LNCS 5086

Kaisa Nyberg (Ed.)

Fast Software Encryption

15th International Workshop, FSE 2008
Lausanne, Switzerland, February 2008
Revised Selected Papers

INTERNATIONAL ASSOCIATION FOR CRYPTOLOGIC RESEARCH

i a c r

Springer

Lecture Notes in Computer Science 5086

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Kaisa Nyberg (Ed.)

Fast Software Encryption

15th International Workshop, FSE 2008
Lausanne, Switzerland, February 10-13, 2008
Revised Selected Papers

Springer

Volume Editor

Kaisa Nyberg
Helsinki University of Technology
Department of Information and Computer Science
Konemiehentie 2, 02150 Espoo, Finland
E-mail: kaisa.nyberg@tkk.fi

Library of Congress Control Number: 2008930931

CR Subject Classification (1998): E.3, I.1

LNCS Sublibrary: SL 4 – Security and Cryptology

ISSN 0302-9743
ISBN-10 3-540-71038-8 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-71038-7 Springer Berlin Heidelberg New York

Springer is a part of Springer Science+Business Media

springer.com

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12435869 06/3180 5 4 3 2 1 0

Preface

Fast Software Encryption (FSE) is the 15th in a series of workshops on symmetric cryptography. It is sponsored by the International Association for Cryptologic Research (IACR), and previous FSE workshops have been held around the world:

1993 Cambridge, UK 1994 Leuven, Belgium 1996 Cambridge, UK
1997 Haifa, Israel 1998 Paris, France 1999 Rome, Italy
2000 New York, USA 2001 Yokohama, Japan 2002 Leuven, Belgium
2003 Lund, Sweden 2004 New Delhi, India 2005 Paris, France
2006 Graz, Austria 2007 Luxembourg, Luxembourg

The FSE workshop is devoted to the foreground research on fast and secure primitives for symmetric cryptography, including the design and analysis of block ciphers, stream ciphers, encryption schemes, analysis and evaluation tools, hash functions, and message authentication codes.

This year 72 papers were submitted to FSE including a large number of high-quality and focused submissions, from which 26 papers for regular presentation and 4 papers for short presentation were selected. I wish to thank the authors of all submissions for their scientific contribution to the workshop. The workshop also featured an invited talk by Lars R. Knudsen with the title "Hash functions and SHA-3." The traditional rump session with short informal presentations on current topics was organized and chaired by Daniel J. Bernstein.

Each submission was reviewed by at least three Program Committee members. Each submission originating from the Program Committee received at least five reviews. The final selection was made after a thorough discussion. I wish to thank all Program Committee members and referees for their generous work. I am also grateful to Thomas Baignères for maintaining and customizing the iChair review management software, which offered an excellent support for the demanding reviewing task. I would also like to thank him for setting up a beautiful and informative website and for compiling the pre-proceedings.

The efforts of the team members of the local Organizing Committee at Lausanne led by Serge Vaudenay and Thomas Baignères were particularly appreciated by the over 200 cryptographers who came from all over the world to attend the workshop. The support given to the FSE 2008 workshop by the sponsors École Polytechnique Fédérale de Lausanne, Nagravision and Nokia is also gratefully acknowledged.

March 2008 Kaisa Nyberg

FSE 2008

February 10–13, 2008, Lausanne, Switzerland

Sponsored by the
International Association for Cryptologic Research (IACR)

Program and General Chairs

Program Chair	Kaisa Nyberg Helsinki University of Technology and NOKIA, Finland
General Co-chairs	Serge Vaudenay and Thomas Baignères École Polytechnique Fédérale de Lausanne, Switzerland

Program Committee

Frederik Armknecht	Ruhr-University Bochum, Germany
Steve Babbage	Vodafone, UK
Alex Biryukov	University of Luxembourg, Luxembourg
John Black	University of Colorado, USA
Anne Canteaut	INRIA, France
Claude Carlet	University of Paris 8, France
Joan Daemen	STMicroelectronics, Belgium
Orr Dunkelman	Katholieke Universiteit Leuven, Belgium
Henri Gilbert	France Telecom, France
Louis Granboulan	EADS, France
Helena Handschuh	Spansion, France
Tetsu Iwata	Nagoya University, Japan
Thomas Johansson	Lund University, Sweden
Antoine Joux	DGA and University of Versailles, France
Pascal Junod	Nagravision, Switzerland
Charanjit Jutla	IBM T.J. Watson Research Center, USA
Mitsuru Matsui	Mitsubishi Electric, Japan
Willi Meier	Fachhochschule Nordwestschweiz, Switzerland
Kaisa Nyberg (Chair)	Helsinki University of Technology and NOKIA, Finland
Elisabeth Oswald	University of Bristol, UK
Josef Pieprzyk	Macquarie University, Australia
Bart Preneel	Katholieke Universiteit Leuven, Belgium
Vincent Rijmen	Katholieke Universiteit Leuven, Belgium and Graz University of Technology, Austria
Greg Rose	Qualcomm, USA

Referees

Jean-Philippe Aumasson
Côme Berbain
Daniel J. Bernstein
Olivier Billet
Nick Bone
Chris Charnes
Joo Yeon Cho
Scott Contini
Jean-Charles Faugère
Martin Feldhofer
Simon Fischer
Ewan Fleischmann
Raphael Fourquet
Thomas Fuhr
Samuel Galice
Sylvain Guilley
Phillip Hawkes
Alexandre Karlov
Shahram Khazaei
Dmitry Khovratovich
Ulrich Kühn
Yann Laigle-Chapuy
Mario Lamberger
Gregor Leander
Marco Macchetti
Stefan Mangard
Stéphane Manuel
Krystian Matusiewicz
Cameron McDonald
Florian Mendel
Marine Minier
Paul Morrisey
Ivica Nikolic
Ludovic Perret
Thomas Peyrin
Duong Hieu Phan
Norbert Pramstaller
Deike Priemuth-Schmid
Emmanuel Prouff
Christian Rechberger
Matthew Robshaw
Markku-Juhani Saarinen
Martin Schläffer
Joern-Marc Schmidt
Yannick Seurin
François-Xavier Standaert
Dirk Stegemann
Jean-Pierre Tillich
Stefan Tillich
Gilles Van Assche
Huaxiong Wang
Ralf-Philipp Weinmann

Sponsors

École Polytechnique Fédérale de Lausanne, Switzerland
Nagravision, Kudelski Group, Switzerland
Nokia, Finland

Table of Contents

SHA Collisions

New Hash Function Designs

Block Cipher Cryptanalysis (I)

Implementation Aspects

Hash Function Cryptanalysis (I)

Stream Cipher Cryptanalysis (I)

Security Bounds

Entropy

Block Cipher Cryptanalysis (II)

Hash Function Cryptanalysis (II)

Stream Cipher Cryptanalysis (II)

Collisions for Step-Reduced SHA-256

Ivica Nikolić* and Alex Biryukov

University of Luxembourg
{ivica.nikolic,alex.biryukov}@uni.lu

Abstract. In this article we find collisions for step-reduced SHA-256. We develop a differential that holds with high probability if the message satisfies certain conditions. We solve the equations that arise from the conditions. Due to the carefully chosen differential and word differences, the message expansion of SHA-256 has little effect on spreading the differences in the words. This helps us to find full collision for 21-step reduced SHA-256, semi-free start collision, i.e. collision for a different initial value, for 23-step reduced SHA-256, and semi-free start near collision (with only 15 bit difference out of 256 bits) for 25-step reduced SHA-256.

1 Introduction

The SHA-2 family of hash functions was introduced to the cryptographic community as a new, more complex, and hopefully, more secure variant of MD4-family of hash functions. The recent results on the widely used MD4-family hash functions SHA-1 and MD5 [6],[7] show flaws in the security of these functions, with respect to collision attacks. The question arises, if the most complex member of MD4-family, the SHA-2 family, is also vulnerable to collision attacks.

Known Results for the SHA-2 Family. Research has been made on finding a local collisions for the SHA-2 family. Gilbert and Handschuh [2] reported a 9-step local collision with probability of the differential path of 2^{-66}. Later, Mendel et al [4] estimated the probability of this local collision to be 2^{-39}. Somitra and Palash obtained a local collision with probability 2^{-42}. Using modular differences Hawkes, Paddon and Rose [3] were able to find a local collision with probability 2^{-39}. As far as we know, the only work on finding a real collision for SHA-2 was made by Mendel et al[4]. They studied message expansion of the SHA-256 and reported a 19-step near collision.

Our Contributions. We find a 9-step differential that holds with probability of $\frac{1}{3}$ by fixing some of the intermediate values and solving the equations that arise. We show that it is not necessary to introduce differences in message words on each step of the differential. This helps us to overcome the message expansion. We use modular substraction differences. Using only one instance of this differential we find 20 and 21-step collisions (collisions for the original initial value) with

* The work of this author was supported by the BFR grant 07/031 of the FNR.

K. Nyberg (Ed.): FSE 2008, LNCS 5086, pp. 1–15, 2008.

probabilities $\frac{1}{3}$ and 2^{-19} respectively. Also, using slightly different differential we were able to find a 23-step semi-free start collision (collisions for a specific initial value) with probability 2^{-21}. Our final result is a 25-step semi-free start near collision with Hamming distance of 15 bits and probability 2^{-34}.

Let $H(M, h_0)$ be a hash function, where M is the input message, and h_0 is the initial chaining value. The following attacks are considered in the paper:

Collision attack: Find messages M_1 and M_2 such that $M_1 \neq M_2$ and $H(M_1, h_0) = H(M_2, h_0)$.

Semi-free start collision attack: Find messages M_1,M_2 and hash value h_0^* such that $M_1 \neq M_2$ and $H(M_1, h_0^*) = H(M_2, h_0^*)$.

Near collision attack: Find messages M_1 and M_2 such that $M_1 \neq M_2$ and Hamming distance between $H(M_1, h_0)$ and $H(M_2, h_0)$ is small compared to the output size n of the hash function.

2 Description of SHA-2

SHA-2 family consists of iterative hash functions SHA-224, SHA-256, SHA-384, and SHA-512. For our purposes, we will describe only SHA-256. The definitions of the rest of the functions can be found in [1]. The SHA-256 takes a message of length less than 2^{64} and produces a 256-bit hash value. First, the input message is padded so the length becomes a multiple of 512, and afterwards each 512-bit message block is processed as an input in the Damgard-Merkle iterative structure. Each iteration calls a compression function which takes for an input a 256-bit chaining value and a 512-bit message block and produces an output 256-bit chaining value. The output chaining value of the previous iteration is an input chaining value for the following iteration. The initial chaining value, i.e. the value for the first iteration, is fixed, and the chaining value produced after the last message block is proceeded is the hash value of the whole message. Internal state of SHA-256 compression function consists of 8 32-bit variables A, B, C, D, E, F, G, and H, each of which is updated on every of the 64 steps. These variables are updated according to the following equations:

$$\begin{aligned}
A_{i+1} &= \Sigma_0(A_i) + Maj(A_i, B_i, C_i) + \Sigma_1(E_i) + Ch(E_i, F_i, G_i) + H_i + K_i + W_i \\
B_{i+1} &= A_i \\
C_{i+1} &= B_i \\
D_{i+1} &= C_i \\
E_{i+1} &= \Sigma_1(E_i) + Ch(E_i, F_i, G_i) + H_i + K_i + W_i + D_i \\
F_{i+1} &= E_i \\
G_{i+1} &= F_i \\
H_{i+1} &= G_i
\end{aligned}$$

The $Maj(X,Y,Z)$ and $Ch(X,Y,Z)$ are bitwise boolean functions defined as:

$$Ch(X,Y,Z) = (X \wedge Y) \vee (\neg X \wedge Z)$$
$$Maj(X,Y,Z) = (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z)$$

For SHA-256 $\Sigma_0(X)$ and $\Sigma_1(X)$ are defined as:

$$\Sigma_0(X) = ROTR^2(X) \oplus ROTR^{13}(X) \oplus ROTR^{22}(X)$$
$$\Sigma_1(X) = ROTR^6(X) \oplus ROTR^{11}(X) \oplus ROTR^{25}(X)$$

State update function uses constants K_i, which are different for every step. The 512-bit message block itself is divided in 16 32-bit bit words: $m_0, m_1, \ldots, m_{16}$. Afterwards, the message block is expanded to 64 32-bit words according to the following rule:

$$W_i = \begin{cases} m_i, & 0 \le i \le 15 \\ \sigma_1(W_{i-2}) + W_{i-7} + \sigma_0(W_{i-15}) + W_{i-16}, & i > 15 \end{cases}$$

For SHA-256 $\sigma_0(X)$ and $\sigma_1(X)$ are defined as:

$$\sigma_0(X) = \ ROTR^7(X) \oplus ROTR^{18}(X) \oplus SHR^3(X)$$
$$\sigma_1(X) = ROTR^{17}(X) \oplus ROTR^{19}(X) \oplus SHR^{10}(X)$$

The compression function after the 64-th step adds the initial values to the chaining variables, i.e. the hash result of the compression function is:

$$h(M) = (A_{64}+A_0, B_{64}+B_0, C_{64}+C_0, D_{64}+D_0, E_{64}+E_0, F_{64}+F_0, G_{64}+G_0, H_{64}+H_0).$$

These values become the initial chaining value for the next compression function.

3 Technique for Creating Collisions

Differences used in this paper are subtractions mod 2^{32} differences.

We use the following notation:

$$\Delta X = X^{'} - X,\ X \in \{A, B, D, D, E, F, G, H, W, m\},$$
$$\Delta Maj^i(\Delta a, \Delta b, \Delta c) = Maj(A_i + \Delta a, B_i + \Delta b, C_i + \Delta c) - Maj(A_i, B_i, C_i),$$
$$\Delta Ch^i(\Delta e, \Delta f, \Delta g) = Ch(E_i + \Delta e, F_i + \Delta f, G_i + \Delta g) - Ch(E_i, F_i, G_i).$$
$$\Delta \Sigma_0(A_i) = \Sigma_0(A_i^{'}) - \Sigma_0(A_i)$$
$$\Delta \Sigma_1(E_i) = \Sigma_1(E_i^{'}) - \Sigma_1(E_i)$$
$$\Delta \sigma_0(m_i) = \sigma_0(m_i^{'}) - \sigma_0(m_i)$$
$$\Delta \sigma_1(m_i) = \sigma_1(m_i^{'}) - \sigma_1(m_i)$$

We introduce perturbation on step i and in the following 8 steps we try to correct the differences in the internal variables. We use the following differential:

Table 1. A 9 step differential for SHA-2 family. Notice that only 5 differences are introduced, i.e. in steps $i, i+1, i+2, i+3$, and $i+8$.

step	ΔA	ΔB	ΔC	ΔD	ΔE	ΔF	ΔG	ΔH	ΔW
i	0	0	0	0	0	0	0	0	1
i+1	1	0	0	0	1	0	0	0	δ_1
i+2	0	1	0	0	-1	1	0	0	δ_2
i+3	0	0	1	0	0	-1	1	0	δ_3
i+4	0	0	0	1	0	0	-1	1	0
i+5	0	0	0	0	1	0	0	-1	0
i+6	0	0	0	0	0	1	0	0	0
i+7	0	0	0	0	0	0	1	0	0
i+8	0	0	0	0	0	0	0	1	δ_4
i+9	0	0	0	0	0	0	0	0	0

As you can see from the table (column ΔW), only the perturbation has been fixed. All the other differences are to be determined.

3.1 Conditions for the Local Collision

From the definition of SHA-2, focusing on registers A_{i+1} and E_{i+1}, we get:

$$\Delta A_{i+1} - \Delta E_{i+1} = \Delta\Sigma_0(A_i) + \Delta Maj^i(\Delta A_i, \Delta B_i, \Delta C_i) - \Delta D_i,$$
$$\Delta E_{i+1} = \Delta\Sigma_1(E_i) + \Delta Ch^i(\Delta E_i, \Delta F_i, \Delta G_i) + \Delta H_i + \Delta D_i + \Delta W_i.$$

We will keep in mind that if $\Delta A_i = \Delta B_i = \Delta C_i = 0$ then $\Delta Maj^i(0,0,0) = 0$. Also if $\Delta E_i = \Delta F_i = \Delta G_i = 0$ then $\Delta Ch^i(0,0,0) = 0$.

We fix the differences for the registers A and E (as shown in the table). The variables B, C, D, F, G, H can only inherit the values from A and E. So, for each step we get some equations with respect to δ_i and A_i or E_i.

Step i+1. We have that $\Delta D_i = 0$, $\Delta H_i = 0$, $\Delta\Sigma_0(A_i) = 0$, $\Delta\Sigma_1(E_i) = 0$. We require $\Delta A_{i+1} = 1$, $\Delta E_{i+1} = 1$. So we deduce:

$$\Delta W_i = 1 \tag{1}$$

Step i+2. We have that $\Delta D_{i+1} = 0$, $\Delta H_{i+1} = 0$. We require $\Delta A_{i+2} = 0$, $\Delta E_{i+2} = -1$. We want also $\Delta\Sigma_0(A_{i+1}) = 1$ to be satisfied. So we deduce:

$$\Delta Maj^{i+1}(1,0,0) = 0, \tag{2}$$
$$\Delta W_{i+1} = -1 - \Delta Ch^{i+1}(1,0,0) - \Delta\Sigma_1(E_{i+1}). \tag{3}$$
$$\Delta\Sigma_0(A_{i+1}) = 1 \tag{4}$$

Step i+3. We have that $\Delta D_{i+2} = 0$, $\Delta H_{i+2} = 0$, $\Delta\Sigma_0(A_{i+2}) = 0$. We require $\Delta A_{i+3} = 0$, $\Delta E_{i+3} = 0$. So we deduce:

$$\Delta Maj^{i+2}(0,1,0) = 0, \tag{5}$$
$$\Delta W_{i+2} = -\Delta\Sigma_1(E_{i+2}) - \Delta Ch^{i+2}(-1,1,0). \tag{6}$$

Step i+4. We have that $\Delta D_{i+3} = 0$, $\Delta H_{i+3} = 0$, $\Delta \Sigma_0(A_{i+3}) = 0$, $\Delta \Sigma_1(E_{i+3}) = 0$. We require $\Delta A_{i+4} = 0$, $\Delta E_{i+4} = 0$. So we deduce:

$$\Delta Maj^{i+3}(0,0,1) = 0, \tag{7}$$

$$\Delta W_{i+3} = -\Delta Ch^{i+3}(0,-1,1). \tag{8}$$

Step i+5. We have that $\Delta D_{i+4} = 1$, $\Delta H_{i+4} = 1$, $\Delta \Sigma_0(A_{i+4}) = 0$, $\Delta \Sigma_1(E_{i+4}) = 0$. We require $\Delta A_{i+5} = 0$, $\Delta E_{i+5} = 1$. So we deduce:

$$\Delta Ch^{i+4}(0,0,-1) = -1. \tag{9}$$

Step i+6. We have that $\Delta D_{i+5} = 0$, $\Delta H_{i+5} = -1$, $\Delta \Sigma_0(A_{i+5}) = 0$. We require $\Delta A_{i+6} = 0$, $\Delta E_{i+6} = 0$. We want also $\Delta \Sigma_0(E_{i+5}) = 1$ to be satisfied. So we deduce:

$$\Delta Ch^{i+5}(1,0,0) = 0. \tag{10}$$

$$\Delta \Sigma_1(E_{i+5}) = 1 \tag{11}$$

Step i+7. We have that $\Delta D_{i+6} = 0$, $\Delta H_{i+6} = 0$, $\Delta \Sigma_0(A_{i+6}) = 0$, $\Delta \Sigma_1(E_{i+6}) = 0$. We require $\Delta A_{i+7} = 0$, $\Delta E_{i+7} = 0$. So we deduce:

$$\Delta Ch^{i+6}(0,1,0) = 0. \tag{12}$$

Step i+8. We have that $\Delta D_{i+7} = 0$, $\Delta H_{i+7} = 0$, $\Delta \Sigma_0(A_{i+7}) = 0$, $\Delta \Sigma_1(E_{i+7}) = 0$. We require $\Delta A_{i+8} = 0$, $\Delta E_{i+8} = 0$. So we deduce:

$$\Delta Ch^{i+7}(0,0,1) = 0. \tag{13}$$

Step i+9. We have that $\Delta D_{i+8} = 0$, $\Delta H_{i+8} = 1$, $\Delta \Sigma_0(A_{i+8}) = 0$, $\Delta \Sigma_1(E_{i+8}) = 0$. We require $\Delta A_{i+9} = 0$, $\Delta E_{i+9} = 0$. So we deduce:

$$\Delta W_{i+8} = -1. \tag{14}$$

3.2 Solution of the System of Equations

Let's first observe (4) and (11). From the differential we can see that $\Delta A_{i+1} = \Delta E_{i+5} = 1$. It means that we want the functions $\Delta \Sigma_0(A_{i+1}), \Delta \Sigma_1(E_{i+5})$ to preserve the difference 1, in other words:

$$\Sigma_0(A_{i+1} + 1) - \Sigma_0(A_{i+1}) = 1,$$
$$\Sigma_1(E_{i+5} + 1) - \Sigma_1(E_{i+5}) = 1.$$

The only solution to these equations is $A_{i+1} = E_{i+5} = -1$, so we get:

$$A_{i+1} = -1, \qquad A^{'}_{i+1} = 0, \tag{15}$$

$$E_{i+5} = -1, \qquad E^{'}_{i+5} = 0. \tag{16}$$

Now let's consider the function $\varDelta Maj^i = Maj(A_i^{'}, B_i^{'}, C_i^{'}) - Maj(A_i, B_i, C_i)$. Let's suppose that $B_i^{'} = B_i$, $C_i^{'} = C_i$ and A_i and $A_i^{'}$ differ in every single bit, i.e. $A_i \oplus A_i^{'} =$`0xffffffff`. Then:

$$\varDelta Maj^i = 0 \Leftrightarrow B_i = C_i$$

Therefore (2) gives us $B_{i+1} = C_{i+1}$, which is $A_i = A_{i-1}$. With the same reasoning we can deduce from (5) that $A_{i+2} = A_i$, and from (7) that $A_{i+3} = A_{i+2}$. So, from (2),(5) and (7) we get that

$$A_{i-1} = A_i = A_{i+2} = A_{i+3} \tag{17}$$

Similarly to what we have done with Maj, now let's consider $\varDelta Ch^i$ and suppose that $F_i^{'} = F_i$, $G_i^{'} = G_i$ and E_i and $E_i^{'}$ differ in every single bit. Then:

$$\varDelta Ch^i = 0 \Leftrightarrow F_i = G_i$$

Therefore (10) and the result (16) gives us $F_{i+5} = G_{i+5}$, which is:

$$E_{i+4} = E_{i+3} \tag{18}$$

Solving (12) requires slightly different reasoning; if we have $E_{i+6} = E_{i+6}^{'}$, $G_{i+6} = G_{i+6}^{'}$ and F_{i+6} and $F_{i+6}^{'}$ would differ in every bit (and they do, see (16)) then :

$$\varDelta Ch^{i+6} = 0 \Leftrightarrow E_{i+6} = 0. \tag{19}$$

Analogously, from (13) we get:

$$E_{i+7} = -1 \tag{20}$$

The only remaining condition is (9):

$$\varDelta Ch^{i+4} = Ch(E_{i+4}, F_{i+4}, G_{i+4}^{'}) - Ch(E_{i+4}, F_{i+4}, G_{i+4}) = -1,\ G_{i+4}^{'} - G_{i+4} = -1.$$

The words $E_{i+4}, F_{i+4}, G_{i+4}$ are already determined to satisfy the previous conditions. So, we don't have any degrees of freedom left to control precisely the solution of this equation. Therefore we will try to find the probability that this condition holds. We can see that it holds if and only if register E_{i+4} has 0's in the bits where $G_{i+4}^{'}$ and G_{i+4} are different. The $G_{i+4}^{'}$ and G_{i+4} can differ in the last i bits, where $1 \leq i \leq 32.$, and these bits are uniquely determined. So, for the probability we get:

$$\sum_{i=1}^{i=32} \text{P}\{\text{Last i bits of } E_{i+4} \text{ are zero}\} \times \text{ P}\{\text{Difference in the exactly i last bits}\} =$$

$$= \sum_{i=1}^{i=32} \frac{1}{2^i}\frac{1}{2^i} \approx \frac{1}{3}.$$

So, the overall probability of our differential is $\frac{1}{3} = 2^{-1.58}$.

The differences in message words of the differential as in Table 1 are the following:

$$\begin{aligned}
\delta_1 &= -1 - \Delta Ch^{i+1}(1,0,0) - \Delta\Sigma_1(E_{i+1}),\\
\delta_2 &= -\Delta\Sigma_1(E_{i+2}) - \Delta Ch^{i+2}(-1,1,0),\\
\delta_3 &= -\Delta Ch^{i+3}(0,-1,1)\\
\delta_4 &= -1
\end{aligned}$$

Notice that the condition (17) shows us that $A_i{=}B_i$ has to hold.

4 Full, Semi-free and Near Collisions for Step-Reduced SHA-256

Our attack technique is the following:

1. Introduce perturbation at step i;
2. Correct the differences in the following 8 steps (probability of success is the probability of our differential, i.e. $\frac{1}{3}$). After the last step of the differential, the differences in the internal variables are zero;
3. All the message words that follow the last step of the differential have to have zero differences;

4.1 20-Step Collision

From the Table 3 of Appendix A we can see that the words m_5, m_6, m_7, m_8, and m_{13} are used only once in the first 20 steps of SHA-2, i.e. they are not used to compute the values of expanded words W_{16}, W_{17}, W_{18}, and W_{19}. This means that message expansion doesn't introduce any difference after the last step of the differential. So, we get collision for 20 step reduced SHA-2, and the collisions can be found practically by hand. The probability of collision is $2^{-1.58}$.

4.2 21-Step Collision

From the Table 3 of Appendix A we can easily see that we have to consider message expansion since there are no message words that are used only once in the first 21 steps and that have the proper indexes for the differential.

We will introduce differences in the words m_6, m_7, m_8, m_9, and m_{14}. The words m_6, m_7, m_8 are used only once in the first 21 steps. Therefore the message expansion in the first 21 steps is irrelevant with respect to these words, i.e. differences in these words don't introduce any other new differences, after the last step of the differential(step 14). Now, we want to find words $m_9, m_9^{'}, m_{14}, m_{14}^{'}$ such that after the 14-th step, the message expansion will not introduce any difference in the following steps. From the Table 3 of Appendix A we can see that the words m_9 and m_{14} are used in W_{16}, W_{18}, and W_{20}. So, from the definition of W_i we get the equations:

$$\Delta W_{16} = \Delta\sigma_1(m_{14}) + \Delta m_9 + \Delta\sigma_0(m_1) + \Delta m_0 = 0 \tag{21}$$

$$\Delta W_{17} = \Delta\sigma_1(m_{15}) + \Delta m_{10} + \Delta\sigma_0(m_2) + \Delta m_1 = 0 \tag{22}$$

$$\Delta W_{18} = \Delta\sigma_1(W_{16}) + \Delta m_{11} + \Delta\sigma_0(m_3) + \Delta m_2 = 0 \tag{23}$$

$$\Delta W_{19} = \Delta\sigma_1(W_{17}) + \Delta m_{12} + \Delta\sigma_0(m_4) + \Delta m_3 = 0 \tag{24}$$

$$\Delta W_{20} = \Delta\sigma_1(W_{18}) + \Delta m_{13} + \Delta\sigma_0(m_5) + \Delta m_4 = 0 \tag{25}$$

Obviously if $m_i^{'} = m_i$ ($W_i^{'} = W_i$) then $\Delta\sigma_0(m_i) = 0$ ($\Delta\sigma_0(W_i) = 0$). This means that $\Delta W_{17} = \Delta W_{19} = 0$. If we can make so that $\Delta W_{16} = 0$ then $\Delta W_{18} = \Delta W_{20} = 0$. So, we get the equation:

$$\Delta\sigma_1(m_{14}) + \Delta m_9 = 0 \tag{26}$$

Considering that $\Delta m_{14} = \delta_4 = -1$, and m_9 can take any value, our experimental results (Monte Carlo method with 2^{32} trials) give us a probability of $2^{-17.5}$ that Δm_{14} and Δm_9 satisfy this equation. Therefore, the overall probability of 21 step collision is around 2^{-19}.

4.3 23-Step Semi-free Start Collision

For 23 step collision we introduce differences in the words m_9, m_{10}, m_{11}, and m_{12}.

If we would follow our differential, we are supposed to introduce difference in the message word W_{17}. We can not control W_{17} directly because it is an expanded word. From the condition $W_{17} = \delta_4 = -1$ (differential) and the message expansion, we get:

$$\Delta W_{17} = \Delta\sigma_1(m_{15}) + \Delta m_{10} + \Delta\sigma_0(m_2) + \Delta m_1 = -1.$$

Since $\Delta m_{15} = \Delta m_2 = \Delta m_1 = 0$, we get:

$$\Delta m_{10} = -1. \tag{27}$$

In our original differential there are no message differences in the word W_{16}. But for W_{16} we have:

$$\Delta W_{16} = \Delta\sigma_1(m_{14}) + \Delta m_9 + \Delta\sigma_0(m_1) + \Delta m_0.$$

Obviously only $\Delta m_9 \neq 0$ and therefore $\Delta W_{16} = \Delta m_9 = 1 \neq 0$. Therefore we shall use slightly different differential:one were there is a difference in the word W_{16}. To keep everything else unchanged, the equations for the step 17 become the following:

$$\Delta E_{17} = \Delta\Sigma_1(E_{16}) + \Delta Ch^{16}(0,0,1) + \Delta D_{16} + \Delta H_{16} + \Delta W_{16}.$$

From the differential we can see that: $\Delta E_{17} = \Delta\Sigma_1(E_{16}) = \Delta D_{16} = \Delta H_{16} = 0$. Therefore we get:

$$\Delta Ch^{16}(0,0,1) + \Delta W_{16} = 0. \tag{28}$$

Now, let's observe the other words of the message expansion.

For W_{18} we have:

$$W_{18} = \Delta\sigma_1(W_{16}) + \Delta m_{11} + \Delta\sigma_0(m_3) + \Delta m_2 = 0$$

Since $\Delta m_3 = \Delta m_2 = 0, \Delta W_{16} = 1$ we get the equation:

$$\Delta\sigma_1(W_{16}) + \Delta m_{11} = 0. \tag{29}$$

For W_{19} we have:

$$W_{19} = \Delta\sigma_1(W_{17}) + \Delta m_{12} + \Delta\sigma_0(m_4) + \Delta m_3 = 0$$

Since $\Delta m_4 = \Delta m_3 = 0, \Delta W_{17} = -1$ we get the equation:

$$\Delta\sigma_1(W_{17}) + \Delta m_{12} = 0. \tag{30}$$

For W_{20} we have:

$$W_{20} = \Delta\sigma_1(W_{18}) + \Delta m_{13} + \Delta\sigma_0(m_5) + \Delta m_4 = 0$$

Since $\Delta W_{18} = \Delta m_{13} = \Delta m_5 = \Delta m_4 = 0$ we get that this equation is satisfied for all values of W_{18}, m_{13}, m_5, m_4.

For W_{21} we have:

$$W_{21} = \Delta\sigma_1(W_{19}) + \Delta m_{14} + \Delta\sigma_0(m_6) + \Delta m_5 = 0$$

Since $\Delta W_{19} = \Delta m_{14} = \Delta m_6 = \Delta m_5 = 0$ we get that this equation is satisfied for all values of W_{19}, m_{14}, m_6, m_5.

For W_{22} we have:

$$W_{22} = \Delta\sigma_1(W_{20}) + \Delta m_{15} + \Delta\sigma_0(m_7) + \Delta m_6 = 0$$

Since $\Delta W_{20} = \Delta m_{15} = \Delta m_7 = \Delta m_6 = 0$ we get that this equation is satisfied for all values of W_{20}, m_{15}, m_7, m_6.

For W_{23} we have:

$$W_{23} = \Delta\sigma_1(W_{21}) + \Delta W_{16} + \Delta\sigma_0(m_8) + \Delta m_7 = 0$$

Since $\Delta W_{21} = \Delta m_8 = \Delta m_7$ and $\Delta W_{16} \neq 0$ we get that this equation has no solution. That is why we can not get more than 23 step collision.

Let's try to solve (27), (28), (29) and (30).

For (27) and the value of the register E_{11} from the differential's conditions we have:

$$\Delta E_{11} = \Delta\Sigma_1(E_{10})) + \Delta Ch^{10}(1,0,0) + \Delta m_{10}.$$

Since $\Delta E_{11} = m_{10} = -1$ we get:

$$\Delta\Sigma_1(E_{10}) + \Delta Ch^{10}(1,0,0) = 0.$$

We solve this equation by setting $\Delta\Sigma_1(E_{10}) = 1$ and $\Delta Ch^{10}(1,0,0) = -1$. The first one has solution:

$$E_{10} = -1,\ E_{10}^{'} = 0. \tag{31}$$

The second equation holds for the values:

$$F_{10} = G_{10} + 1. \tag{32}$$

Now let's turn to the solution of (28). Using the fact that $G_{16} = -1$ and $G_{16}^{'} = 0$, we get that this equation is satisfied if:

$$E_{16} = \texttt{0xfffffffe} \tag{33}$$

Let's observe the equation (30). From the conditions of the differential we have:

$$\Delta E_{13} = \Delta\Sigma_1(E_{12}) + \Delta Ch^{12}(0,-1,1) + \Delta H_{12} + \Delta D_{12} + \Delta m_{12}$$

Since $\Delta E_{13} = \Delta E_{12} = \Delta H_{12} = \Delta D_{12} = 0$ we get:

$$\Delta Ch^{12}(0,-1,1) + \Delta m_{12} = 0.$$

If we substitute m_{12} from (30) we can get:

$$\Delta Ch^{12}(0,-1,1) = \Delta\sigma_1(-1).$$

This equation can be satisfied if we can control E_{12} and F_{12}.

For E_{12}, from the definition of A_{12} and E_{12} we have:

$$A_{12} - E_{12} = \Sigma_1(A_{11}) + Ch(A_{11}, B_{11}, C_{11}) - D_{11}$$

Considering that $A_{12} = A_{11} = C_{11} = D_{11}$ from the differential's conditions, we get:

$$E_{12} = A_9 - \Sigma_1(A_9)$$

Since A_9 can take any value (we consider semi-free start collision) we deduce that E_{12} can take any value.

The F_{12} value, which is E_{11} can be controlled through H_{10}. Notice that changing H_{10}, which is G_9, doesn't effect E_{10}, because from (31) we can see that E_{10} always takes the arranged value.

We proved that we can fully control E_{12} and F_{12}. We can choose some specific value for $\Delta\sigma_1(-1)$ which is possible to get from $\Delta Ch^{12}(0,-1,1)$, and set the A_9 and G_9 so that the equation (30) will hold.

The last equation, i.e. (29), is satisfied for some specific values of W_{16} and m_{11}. Our experimental results show that with probability $2^{-19.5}$ W_{16} and m_{11} satisfy (29). Therefore the overall probability of semi-free start collision for 23-step reduced SHA-256 is around 2^{-21}.

4.4 25-Steps Semi-free Start Near Collision

Let's suppose we have a semi-free start collision on the 23-rd step. Each following step introduces differences in the chaining variables A and E. The variables B, C, D, F, G, H can only inherit differences from A and E. Therefore, for each step, we should try to minimize the differences in A and E. When we say to minimize the differences we mean to minimize the Hamming distances between $A^{'}$ and A, and between $E^{'}$ and E.

Step 24

$$\min_{W'_{23}-W_{23}=1} h_d(E^{'}_{24}, E_{24}) = \min_{W'_{23}-W_{23}=1} h_d(C_1+1, C_1) = 1,$$

where $C_1 = \Sigma_1(E_{23}) + Ch(E_{23}, F_{23}, G_{23}) + H_{23} + D_{23} + K_{23} + W_{23}$.

$$\min_{W'_{23}-W_{23}=1} h_d(A^{'}_{24}, A_{24}) = \min_{W'_{23}-W_{23}=1} h_d(C_2+1, C_2) = 1,$$

where $C_2 = \Sigma_0(A_{23})+Maj(A_{23}, B_{23}, C_{23})+\Sigma_1(E_{23})+Ch(E_{23}, F_{23}, G_{23})+H_{23}+K_{23}+W_{23}$.

We have the minimal Hamming distances when $C_1^{32} = C_2^{32} = 0$, which means with probability 2^{-2}.

Step 25

$$\min_{W'_{24}-W_{24}=-1+\Delta\sigma_0(1)} h_d(E^{'}_{25}, E_{25}) =$$

$$= \min h_d(\Sigma_1(E^{'}_{24})+Ch(E^{'}_{24}, F_{24}, G_{24})-1+\sigma_0(m_9+1)+C_1, \Sigma_1(E_{24})+Ch(E_{24}, F_{24}, G_{24})+\sigma_0(m_9)+C_1),$$

where $C_1 = H_{24}+D_{24}+K_{24}+\sigma_1(W_{22})+m_8$. If $F_{24}^{32} = 1$ and $G_{24}^{32} = 0$ (probability 2^{-2}) then, considering that $E_{24}^{'32} = 1, E_{24}^{32} = 0$, we have $Ch(E^{'}_{24}, F_{24}, G_{24}) - 1 = Ch(E_{24}, F_{24}, G_{24})$, and we can rewrite the last expression as:

$$\min h_d(\Sigma_1(E^{'}_{24}) + \sigma_0(m_9+1) + C_2, \Sigma_1(E_{24}) + \sigma_0(m_9) + C_2),$$

where $C_2 = C_1 + Ch(E_{24}, F_{24}, G_{24})$.

If no carry occurs due to the differences, then the above minimum is:

$$\min h_d(\Sigma_1(E^{'}_{24}) + \sigma_0(m_9+1) + C_2, \Sigma_1(E_{24}) + \sigma_0(m_9) + C_2) = 5.$$

For $\Sigma_1(E^{'}_{24})$ (difference in three bits) there are no carries with probability 2^{-3}. For $\sigma_0(m_9+1)$ (two differences if $m_9^{32} = 0$) with probability 2^{-3}. Therefore the minimum is 5 with probability 2^{-8}.

Using the same methods we can get:

$$\min_{W'_{24}-W_{24}=-1+\Delta\sigma_0(1)} h_d(A^{'}_{25}, A_{25}) = 8,$$

with probability 2^{-11}. Notice that if minimum holds for A_{25} then it holds for E_{25}.

So, for the whole hash value, we have:

$$h_d((A_{25}^{'}, B_{25}^{'}, C_{25}^{'}, D_{25}^{'}, E_{25}^{'}, F_{25}^{'}, G_{25}^{'}, H_{25}^{'})), (A_{25}, B_{25}, C_{25}, D_{25}, E_{25}, F_{25}, G_{25}, H_{25})) =$$
$$= h_d((A_{25}^{'}, A_{24}^{'}, C_{25}, D_{25}, E_{25}^{'}, E_{24}^{'}, G_{25}, H_{25}), (A_{25}, A_{24}, C_{25}, D_{25}, E_{25}, E_{24}, G_{25}, H_{25})) =$$
$$= h_d(A_{25}^{'}, A_{25}) + h_d(E_{25}^{'}, E_{25}) + h_d(A_{24}^{'}, A_{24}) + h_d(E_{24}^{'}, E_{24}) =$$
$$= 8 + 5 + 1 + 1 = 15$$

Therefore we get a 25-step semi-free start near collision with the Hamming weight of 15 bits and probability 2^{-34}. Notice that we haven't investigated all the possible outcomes of the carry effects. Therefore, it is possible that the real probability is higher.

Table 2. Collision search attacks for SHA-256

# of steps	Type of collision	Complexity(*)	Paper
19	Near collision	(**)	[4]
20	Collision	$2^{1.58}$	This paper
21	Collision	2^{19}	This paper
22	Pseudo-collision	(**)	[4]
23	Semi-free start collision	2^{21}	This paper
25	Semi-free start near collision	2^{34}	This paper

(*) Complexity is measured in reduced SHA-256 calls
(**) Complexity not mentioned in the paper

5 Conclusion

We created a 9-step differential for SHA-256 that holds with high probability. Using the characteristics of this differential, precisely, the fact that not all of the input message words have differences, we were able to overcome the beginning steps of the message expansion. We created a full collisions for 20 and 21-step reduced SHA-256. Also, we found a 23-step reduced semi-free start collision, and 25-step reduced near collision with Hamming distance of 17 out of 256 bits. The complexities of these collisions search attacks are showed in Table 2. Obviously, our results hold for SHA-224 too. For SHA-384 and SHA-512 different equations arise. We have not analyzed them, but our guess is that complexities of the attacks should stay the same.

References

1. Secure Hash Standard. Federal Information Processing Starndard Publication 180-2. U.S. Department of Commerce, National Institute of Standards and Technology (NIST) (2004)
2. Gilbert, H., Handschuh, H.: Security analysis of SHA-256 and sisters. In: Matsui, M., Zuccherato, R.J. (eds.) Selected Areas in Cryptography, 2003. LNCS, vol. 3006, pp. 175–193. Springer, Heidelberg (2003)

3. Hawkes, P., Paddon, M., Rose, G.G.: On Corrective Patterns for the SHA-2 Family. Cryptology eprint Archive (August 2004), http://eprint.iacr.org/2004/207
4. Mendel, F., Pramstaller, N., Rechberger, C., Rijmen, V.: Analysis of step-reduced SHA-256. In: Robshaw, M.J.B. (ed.) FSE 2006. LNCS, vol. 4047, pp. 126–143. Springer, Heidelberg (2006)
5. Sanadhya, S.K., Sarkar, P.: New Local Collision for the SHA-2 Hash Family.Cryptology eprint Archive (2007), http://eprint.iacr.org/2007/352
6. Wang, X., Yin, Y.L., Yu, H.: Finding collisions in the full SHA-1. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 17–36. Springer, Heidelberg (2005)
7. Wang, X., Yu, H.: How to break MD5 and other hash functions. In: Cramer, R. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 19–35. Springer, Heidelberg (2005)

A Message Expansion

Table 3. Message expansion of SHA-2. There is 'x' in the intersection of row with index i and column with index j if W_i uses m_j.

W	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	x															
1		x														
2			x													
3				x												
4					x											
5						x										
6							x									
7								x								
8									x							
9										x						
10											x					
11												x				
12													x			
13														x		
14															x	
15																x
16	x	x								x					x	
17		x	x								x					x
18	x	x	x	x						x		x			x	
19		x	x	x	x						x		x			x
20	x	x	x	x	x	x				x		x		x	x	
21		x	x	x	x	x	x				x		x		x	x
22	x	x	x	x	x	x	x	x		x		x		x	x	x

B Conditions for Collision

Table 4. The differences propagation for 20, 21, and 23-step collisions for SHA-256. Notice that for each collision initial difference is introduced in different steps (steps 5,6,9 respectively).

20 step	21 step	23 step	ΔA	ΔB	ΔC	ΔD	ΔE	ΔF	ΔG	ΔH	ΔW
5	6	9	0	0	0	0	0	0	0	0	1
6	7	10	1	0	0	0	1	0	0	0	δ_1
7	8	11	0	1	0	0	-1	1	0	0	δ_2
8	9	12	0	0	1	0	0	-1	1	0	δ_3
9	10	13	0	0	0	1	0	0	-1	1	0
10	11	14	0	0	0	0	1	0	0	-1	0
11	12	15	0	0	0	0	0	1	0	0	0
12	13	16	0	0	0	0	0	0	1	0	δ_5
13	14	17	0	0	0	0	0	0	0	1	-1
14	15	18	0	0	0	0	0	0	0	0	0

Table 5. The values of the word differences in 20, 21, and 23-step collisions for SHA-256. Notice that 23-step semi-free start collision has a word difference in δ_5. That is why its collision path is slightly different than the one used for 20 and 21-step collision.

	δ_1	δ_2	δ_3	δ_5
20-step	$-1 - \Delta Ch^6(1,0,0) - \Delta\Sigma_1(E_6)$	$-\Delta\Sigma_1(E_7) - \Delta Ch^7(-1,1,0)$	$-\Delta Ch^8(0,-1,1)$	0
21-step	$-1 - \Delta Ch^7(1,0,0) - \Delta\Sigma_1(E_7)$	$-\Delta\Sigma_1(E_8) - \Delta Ch^8(-1,1,0)$	$-\Delta Ch^9(0,-1,1)$	0
23-step	-1	$-\Delta\Sigma_1(E_{11}) - \Delta Ch^{11}(-1,1,0)$	$-\Delta Ch^{12}(0,-1,1)$	1

Table 6. The additional conditions that have to hold in order to get a 20, 21, and 23-step collisions for SHA-256

20-step	$A_4 = A_5 = A_7 = A_8$ $A_6 = -1, A'_6 = 0$	$E_9 = E_8, E_{10} = -1, E'_{10} = 0$ $E_{11} = 0, E_{12} = -1$	$\Delta Ch^9(0,0,-1) = -1$
21-step	$A_5 = A_6 = A_8 = A_9$ $A_7 = -1, A'_7 = 0$	$E_{10} = E_9, E_{11} = -1, E'_{11} = 0$ $E_{12} = 0, E_{13} = -1$	$\Delta Ch^{10}(0,0,-1) = -1$ $\Delta\sigma_1(-1) + \delta_3 = 0$
23-step	$A_8 = A_6 = A_9 = A_{10}$ $A_{10} = -1, A'_{10} = 0$	$E_{13} = E_{12}, E_{14} = -1, E'_{14} = 0$ $E_{15} = 0, E_{16} =$`0xfffffffe` $E_9 = E_8 + 1, E_{10} = -1, E'_{10} = 0$	$\Delta Ch^{13}(0,0,-1) = -1$ $\Delta\sigma_1(-1) + \delta_3 = 0$ $\Delta\sigma_1(1) + \delta_2 = 0$

C Collision Examples

Table 7. A 21-step collision for SHA-256

M_0	0004024f	00000000	00000000	00000000	00000000	2c51fd8d	b83daf3**c**	bc852709
	ae18a3e7	1d11dbc7	21d06175	ab551b5f	a48e9a8b	00000000	19000000	00000000
$M_0^{'}$	0004024f	00000000	00000000	00000000	00000000	2c51fd8d	b83daf3d	7c652ab7
	b238a344	1d11dac8	21d06175	ab551b5f	a48e9a8b	00000000	18ffffff	00000000
H	73f5fcd2	682f578e	8d9c3d05	f93ad865	662b0636	a5a5d4c2	32091775	04ac6dae

Table 8. A 23-step semi-free start collision for SHA-256

H_0	cb518aaa	55d8f4ad	231e476a	89ac8889	f29c30cc	2e1f63c5	cf4f2366	75367200
M_0	b5c16a2d	6da1708b	00000000	00000000	00000000	00000000	00000000	00000000
	00000000	a9d5faeb	54eb8149	085be1ce	b9e61e60	9380ae01	efa5a517	cdc5da00
$M_0^{'}$	b5c16a2d	6da1708b	00000000	00000000	00000000	00000000	00000000	00000000
	00000000	a9d5faec	54eb8148	085c0205	b9e61d61	9380ae01	efa5a517	cdc5da00
H	6682cc14	9c825293	bc17ea6d	d89770cf	a69ac7ed	cfa5ee3e	e35c0091	7249d71e

Table 9. A 25-step semi-free start near collision with Hamming distance of 17 bits for SHA-256

H_0	8e204f9e	bca27aea	42da63d7	00f2f219	fd1db715	6389ae13	c6f57538	de4e655c
M_0	c63714eb	13d5fa9c	00000000	00000000	00000000	00000000	00000000	00000000
	00000000	d51b4dba	aeb6f738	61dce9b7	0ab5c01a	83406f01	df65666b	cdc5da00
$M_0^{'}$	c63714eb	13d5fa9c	00000000	00000000	00000000	00000000	00000000	00000000
	00000000	d51b4dbb	aeb6f737	71dd499a	0ab5bf1b	83406f01	df65666b	cdc5da00
H	2e2fcb73	8192d3a4	f85b5a7d	801c4583	9307e51c	cf57fb61	11c48b0d	7131ccd2
$H^{'}$	6c478ef3	8192d3a5	f85b5a7d	801c4583	9127a49c	cf57fb62	11c48b0d	7131ccd2

Collisions on SHA-0 in One Hour

Stéphane Manuel[1,⋆] and Thomas Peyrin[2,3,4,⋆⋆]

[1] INRIA
stephane.manuel@inria.fr
[2] Orange Labs
thomas.peyrin@orange-ftgroup.com
[3] AIST
[4] Université de Versailles Saint-Quentin-en-Yvelines

Abstract. At Crypto 2007, Joux and Peyrin showed that the *boomerang attack*, a classical tool in block cipher cryptanalysis, can also be very useful when analyzing hash functions. They applied their new theoretical results to `SHA-1` and provided new improvements for the cryptanalysis of this algorithm. In this paper, we concentrate on the case of `SHA-0`. First, we show that the previous perturbation vectors used in all known attacks are not optimal and we provide a new 2-block one. The problem of the possible existence of message modifications for this vector is tackled by the utilization of auxiliary differentials from the boomerang attack, relatively simple to use. Finally, we are able to produce the best collision attack against `SHA-0` so far, with a measured complexity of $2^{33,6}$ hash function calls. Finding one collision for `SHA-0` takes us approximatively one hour of computation on an average PC.

Keywords: hash functions, `SHA-0`, boomerang attack.

1 Introduction

Cryptographic hash functions are an important tool in cryptography. Basically, a cryptographic hash function H takes an input of variable size and returns a hash value of fixed length while satisfying the properties of preimage resistance, second preimage resistance, and collision resistance [11]. For a secure hash function that gives an n-bit output, compromising these properties should require 2^n, 2^n, and $2^{n/2}$ operations respectively.

Usually, hash functions are built upon two components: a *compression function* and a *domain extension algorithm*. The former has the same security requirements that a hash function but takes fixed length inputs. The latter defines how to use the compression function in order to handle arbitrary length inputs. From the early beginning of hash functions in cryptography, designers relied on the pioneering work of Merkle and Damgård [8,17] concerning the domain extension

[⋆] The first author is supported in part by the french Agence Nationale de la Recherche under the project designation EDHASH, DPg/ANR-CI FA/VB 2007-010.

[⋆⋆] The second author is supported by the Japan Society for Promotion of Science and the French RNRT SAPHIR project (http://www.crypto-hash.fr).

K. Nyberg (Ed.): FSE 2008, LNCS 5086, pp. 16–35, 2008.

algorithm. Given a collision resistant compression function, it became easy to build a collision resistant hash function. However, it has been recently shown that this iterative process presents flaws [9,12,13,15] and some new algorithms [1,4] with better security properties have been proposed. One can distinguish three different methods for compression function designs: block cipher based, related to a well studied hard problem and from scratch.

The most famous design principle for dedicated hash functions is indisputably the MD-SHA family, firstly introduced by R. Rivest with MD4 [24] in 1990 and its improved version MD5 [23] in 1991. Two years after, the NIST publishes [19] a very similar hash function, SHA-0, that will be patched [20] in 1995 to give birth to SHA-1. This family is still very active, as NIST recently proposed [21] a 256-bit new version SHA-256 in order to anticipate the potential cryptanalysis results and also to increase its security with regard to the fast growth of the computation power. All those hash functions use the Merkle-Damgård extension domain and their compression function, even if considered conceived from scratch, is built upon a dedicated block cipher in Davies-Meyer mode: the output of the compression function is the output of the block cipher with a feed-forward of the chaining variable.

Dobbertin [10] provided the first cryptanalysis of a member of this family with a collision attack against MD4. Later, Chabaud-Joux [7] published the first theoretical collision attack against SHA-0 and Biham-Chen [2] introduced the idea of neutral bits, which led to the computation of a real collision with four blocks of message [3]. Then, a novel framework of collision attack, using modular difference and message modification techniques, surprised the cryptography community [26,27,28,29]. Those devastating attacks broke a lot of hash functions, such as MD4, MD5, SHA-0, SHA-1, RIPEMD or HAVAL-128. In the case of SHA-0 the overall complexity of the attack was 2^{39} message modification processes. Recently, Naito *et al.* [18] lower this complexity down to 2^{36} operations, but we argue in this paper that it is a theoretical complexity and not a measured one.

At Crypto 2007, Joux and Peyrin [14] published a generalization of neutral bits and message modification techniques and applied their results to SHA-1. The so-called *boomerang attack* was first devoted for block ciphers cryptanalysis [25] but their work showed that it can also be used in the hash functions setting. Used in parallel with the automated tool from De Cannière and Rechberger [5] that generates non-linear part of a differential path, this method turns out to be quite easy to use and handy for compression functions cryptanalysis.

This article presents a new attack against the collision resistance of SHA-0 requiring only 2^{33} hash computations and the theoretical analysis is confirmed by experimentation. First, we show that the previously used perturbation vector, originally found by Wang *et al.*, is not optimal. We therefore introduce a new vector, allowing ourselves to use two iterations of the compression function. In order to compensate the loss of the known message modifications due to the perturbation vector change, we use the boomerang attack framework in order to accelerate the collision search. Finally, this work leads to the best collision attack against SHA-0 from now on, now requiring only one hour of computation on an average PC.

We organized the paper as follows. In Section 2, we recall the previous attacks and cryptanalysis techniques for SHA-0. Then, in Section 3, we analyze the perturbation vector problem and give new ones that greatly improve the complexity of previous attacks. We then apply the boomerang technique as a speedup technique in Section 4 and provide the final attack along with its complexity analysis in Section 5. Finally, we draw conclusions in Section 6.

2 Previous Collision Attacks on SHA-0

2.1 A Short Description of SHA-0

SHA-0 [19], is a 160-bit dedicated hash function based on the design principle of MD4. It applies the Merkle-Damgård paradigm to a dedicated compression function. The input message is padded and split into k 512-bit message blocks. At each iteration of the compression function h, a 160-bit chaining variable H_t is updated using one message block M_{t+1}, *i.e* $H_{t+1} = h(H_t, M_{t+1})$. The initial value H_0 (also called IV) is predefined and H_k is the output of the hash function.

The SHA-0 compression function is build upon the Davis-Meyer construction. It uses a function E as a block cipher with H_t for the message input and M_{t+1} for the key input, a feed-forward is then needed in order to break the invertibility of the process:

$$H_{t+1} = E(H_t, M_{t+1}) \boxplus H_t,$$

where $\boxplus$ denotes the addition modulo 2^{32} 32-bit words by 32-bit words. This function is composed of 80 steps (4 rounds of 20 steps), each processing a 32-bit message word W_i to update 5 32-bit internal registers (A, B, C, D, E). The feed-forward consists in adding modulo 2^{32} the initial state with the final state of each register. Since more message bits than available are utilized, a message expansion is therefore defined.

Message Expansion. First, the message block M_t is split into 16 32-bit words $W_0, \ldots, W_{15}$. These 16 words are then expanded linearly, as follows:

$$W_i = W_{i-16} \oplus W_{i-14} \oplus W_{i-8} \oplus W_{i-3} \text{ for } 16 \leq i \leq 79.$$

State Update. First, the chaining variable H_t is divided into 5 32-bit words to fill the 5 registers $(A_0, B_0, C_0, D_0, E_0)$. Then the following transformation is applied 80 times:

$$STEP_{i+1} := \begin{cases} A_{i+1} = (A_i \lll 5) + f_i(B_i, C_i, D_i) + E_i + K_i + W_i, \\ B_{i+1} = A_i, \\ C_{i+1} = B_i \ggg 2, \\ D_{i+1} = C_i, \\ E_{i+1} = D_i. \end{cases}$$

where K_i are predetermined constants and f_i are boolean functions defined in Table 1.

Feed-Forward. The sums modulo 2^{32}: $(A_0 + A_{80})$, $(B_0 + B_{80})$, $(C_0 + C_{80})$, $(D_0 + D_{80})$, $(E_0 + E_{80})$ are concatenated to form the chaining variable H_{t+1}.

Note that all updated registers but A_{i+1} are just rotated copies, so we only need to consider the register A at each step. Thus, we have:

$$A_{i+1} = (A_i \lll 5) + f_i(A_{i-1}, A_{i-2} \ggg 2, A_{i-3} \ggg 2) + A_{i-4} \ggg 2 + K_i + W_i. \quad (1)$$

2.2 First Attacks on SHA-0

The first published attack on SHA-0 has been proposed by Chabaud and Joux in 1998 [7]. It focused on finding linear differential paths composed of interleaved 6-step *local collisions*, which have probability 1 to hold in a linearized version of SHA-0. However, in the standard version of SHA-0, a local collision only has a certain probability to hold. The overall probability of success of the attack is the product of the holding probability of each local collision.

The core of the differential path is represented by a perturbation vector which indicates where the 6-step local collisions are initiated. The probability of success of the attack is then related to the number of local collisions appearing in the perturbation vector. In their paper, Chabaud and Joux have defined 3 necessary conditions on perturbation vectors in order to permit the differential path to end with a collision for the 80-step compression function. Such a perturbation vector should not have truncated local collisions, should not have two consecutive local collisions initiated in the first 16*th* steps and should not start a local collision after step 74. Under these constraints they were able to find a perturbation vector (so-called L-characteristic) with a probability of success of 2^{68}. The running complexity of their attack is decreased to 2^{61} by a careful implementation of the collision search. As the attacker has full control on the first 16 message blocks, those blocks are chosen such that the local collisions of those early steps hold with probability 1. See [7] for more details.

In 2004, Biham and Chen have improved the attack of Chabaud and Joux by introducing the neutral bit technique. The idea is to multiply the number of conformant message pairs up to a certain step s (message pairs that verify the main differential path up to step s) for a cost almost null. This is done by looking for different sets of small modifications in the message words such that each set will have very low impact on the conformance of the message pair up to step s. Basically, the attacker can effectively start the collision search at a higher step than in a normal setting, and this improvement finally led to the

Table 1. Boolean functions and constants in SHA-0

round	step i	$f_i(B,C,D)$	K_i
1	$1 \leq i \leq 20$	$f_{IF} = (B \wedge C) \oplus (\overline{B} \wedge D)$	0x5a827999
2	$21 \leq i \leq 40$	$f_{XOR} = B \oplus C \oplus D$	0x6ed6eba1
3	$41 \leq i \leq 60$	$f_{MAJ} = (B \wedge C) \oplus (B \wedge D) \oplus (C \wedge D)$	0x8fabbcdc
4	$61 \leq i \leq 80$	$f_{XOR} = B \oplus C \oplus D$	0xca62c1d6

computation of the first real collision for SHA-0 with four blocks of message [3] with an overall complexity of 2^{51} functions calls.

2.3 The Wang Approach

The attack on SHA-0 of Wang *et al.* is derived from the approach of Chabaud and Joux. The principle of this attack consists in relaxing two of the three conditions on the perturbation vectors defined by Chabaud and Joux, namely no truncated local collision allowed and no consecutive local collisions in the 16*th* first steps. Relaxing those conditions permits to search for better perturbation vectors, *i.e.* higher probability linear differential paths.

However, the main drawback of this approach is that non corrected perturbations inherited from truncated local collisions appear in the first steps. In order to offset these unwanted perturbations, they had to construct a non linear differential path (so-called NL-characteristic) which connects to the desired linear differential path. Said in other words, they kept the same linear differential mask on the message, but computed a new and much more complex differential mask on the registers for the early steps of SHA-0. A NL-characteristic presents also the advantage that consecutive local collisions in the early steps are no more a problem. Using modular subtraction as the differential, the carry effect (a property of the powers of 2, i.e. $2^j = -2^j - 2^{j+1} \ldots - 2^{j+k-1} + 2^{j+k}$) and by carefully controlling the non linearity of the round function IF, they succeeded to build their NL-characteristic by hand. A NL-characteristic holds only if specific conditions are verified step by step by the register values. In their paper, Wang *et al.* denoted these conditions as *sufficient conditions*. These sufficient conditions are described with respect to the register A into one general form $A_{i,j} = v$ where $A_{i,j}$ denotes the value of bit j of the register A at the step i and where v is a bit value fixed to be 0 or 1 or a value that has been computed before step i.

The NL-characteristic found presents conditions on the initial value of the registers[1]. However, since the initial value is fixed, Wang *et al.* have build their collision with two blocks of message. The first block is needed in order to obtain a chaining variable verifying the conditions, inherited from the NL-characteristic, on the initial values of the register. This is detailed in Figure 1.

The attack of Wang *et al.* is thus divided into two phases. The first one is the pre-computation phase:

1. search for a higher probability L-characteristic by relaxing conditions on the perturbation vectors,
2. build a NL-characteristic which connects to the L-characteristic by offsetting unwanted perturbations,
3. find a first block of message from which the incoming chaining variable verifies the conditions inherited from the NL-characteristic.

[1] The conditions given by Wang *et al.* in their article are incomplete. In fact, two more conditions need to be verified [16,18].

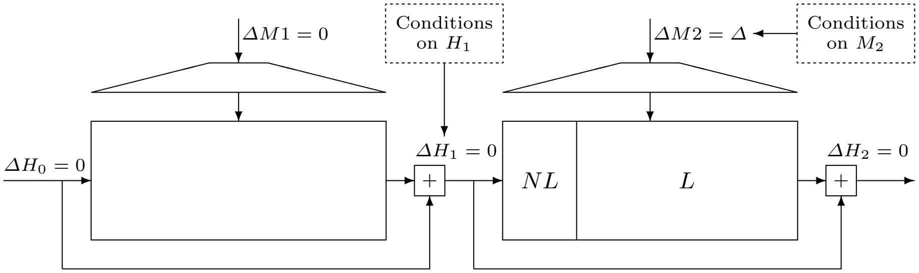


Fig. 1. Attack of Wang *et al*

The second phase is the collision search phase. It consists in searching for a second block of message for which the sufficient conditions on the register values are fulfilled for a maximum number of steps. In order to achieve that goal, they use both *basic modification technique* and *advanced modification technique.* The main idea of the former is simply to set $A_{i,j}$ to the correct bit by modifying the corresponding bit of the message word W_{i-1}. This is only possible for the first 16 steps, where the attacker has full control on the values W_i. The advanced modification technique are to be applied to steps 17 and higher, where the message words W_i are generated by the message expansion. The idea is to modify the message words of previous steps in order to fulfill a condition in a given step. Wang *et al.* claimed in their article that using both basic modification and advanced modification techniques, they are able to fulfill all the sufficient conditions up to step 20. However very few details can be found on advanced modification technique in their article. Finally, their attack has a claimed complexity of 2^{39} SHA-0 operations.

Remark. Wang *et al.* optimized the choice of their perturbation vector taking into account their ability to fulfill the conditions up to step 20.

2.4 Naito *et al.*

Naito *et al.* recently proposed [18] a new advanced modification technique so-called *submarine modification.* Its purpose is to ensure that the sufficient conditions from steps 21 to 24 are fulfilled. The main idea of submarine modifications is to find *modification* characteristics which will permit to manipulate bit values of registers and message words after step 16. Each parallel characteristic is specifically built to satisfy one target condition. In order to construct such a characteristic, Naito *et al.* use two different approaches the *cancel method* and the *transmission method.* The former is based on the local collision principle. Whereas the transmission method combines the recurrence properties of the message expansion and of the step update transformation.

Those modification characteristics define new sets of conditions on register values and message words. The new conditions should not interfere with the already pre-computed sufficient conditions. Naito *et al.* detailed their submarine modifications up to step 17 (see [18] proof of Theorem 1). They remarked that the probability that one of these modifications can satisfy a target condition without affecting the other sufficient conditions is almost 1. No detail is given about the impact of the submarine modifications after step 24.

The claimed complexity of the attack described in [18] is 2^{36} SHA-0 operations. This is a theoretical complexity that will be further discussed in section 5. The given collision example is based on the same NL-characteristic and perturbation vector that Wang *et al.* used to produce their own collision. Taking into account that their submarine modifications permit to fulfill the sufficient conditions up to step 24, Naito *et al.* proposed a new perturbation vector which would therefore minimizes the complexity of the attack. However, in order to effectively build an attack based on the proposed vector, a new NL-characteristic and new submarine modifications should be found.

3 A New Perturbation Vector

In order to lower the complexity of a collision search on SHA-0, high probability L-characteristics are needed. In the previous attacks on SHA-0, the authors have proposed perturbation vectors which do not have local collisions starting after step 74. By relaxing this last condition, it may be possible to find better perturbation vectors. Note that those vectors do not seem to be eligible for a collision search, since they would lead to a near-collision (two compression function outputs with very few bits of difference) instead of a collision. However, this problem can be tackled by using the *multi-block technique* as in [3]: the attacker can take advantage of the feed-forward operation inherited from the Davis-Meyer construction used in the compression function of SHA-0. Said in other words, we allow ourselves to use several message blocks with differences, whereas the previous known attacks on SHA-0 only use one of such blocks of message. Thanks to the new automatic tool from De Cannière and Rechberger [5] that can generate NL-characteristics on SHA-1, computing non linear parts for SHA-0 is relatively easy. Indeed, SHA-1 and SHA-0 only differ on a rotation in the message expansion, which has no effect on the validity of this tool. Moreover, the ability to generate NL-characteristics reduce the multi-block problem to the use of only two blocks. More precisely, we start with a L-characteristic L_1 (defined by a new perturbation vector), and modified on the early steps by a generated NL-characteristic NL_1. We thus obtain a specific near-collision $\Delta H_1 = +d$ after this first block. We then apply the same L-characteristic modified on the early steps by a second generated NL-characteristic NL_2, that takes in account the new incoming chaining variable H_1. Finally, before the feed-forward on this second block, we look for the opposite difference $\Delta E(H_1, m_1) = -d$ and the two differences cancel each over $\Delta H_2 = 0$. This is detailed in Figure 2.

Now that all the conditions on the perturbation vectors are relaxed, we need to define what are the criteria for good perturbation vectors. In order to fulfill the sufficient conditions inherited from NL_1, NL_2 and L_1, we will use basic

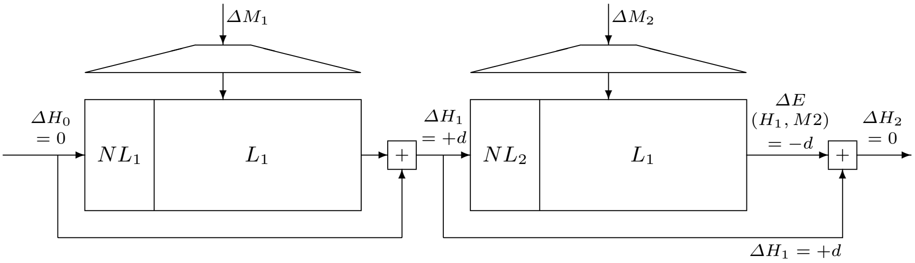


Fig. 2. Multi-block collision on SHA-0

Table 2. A new perturbation vector for SHA-0, along with the number of conditions at each steps (the conditions before step 16 have been removed since not involved in the complexity during the collision search)

Steps 1 to 40	
vector	1 1 1 1 0 1 0 1 1 0 1 1 1 0 0 0 1 0 0 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0
♯ conditions	- - - - - - - - - - - - - - - - 2 0 2 1 1 0 2 0 2 0 1 0 0 0 0 0 0 0 1 0 1 1 0 1

Steps 41 to 80	
vector	0 0 1 0 0 0 0 0 1 0 0 0 0 1 0 0 1 0 1 1 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 1 0 0 0 0
♯ conditions	1 1 1 0 1 1 1 0 1 0 1 1 1 1 0 1 2 1 2 2 1 1 0 0 0 0 1 0 1 0 1 0 1 0 0 1 0 1 0 0

message modifications and boomerang techniques. In consequence, we focused our search on perturbation vectors that have the smallest number of sufficient conditions to fulfill in steps 17 to 80. Namely, a characteristic L_1 for which the probability of success of the attack is maximized. We used the same approach of Chabaud and Joux in order to evaluate the probability of holding of each local collision involved. See [7] and particularly Section 2.2 (Tables 4 and 5) for detailed examples. There are a lot of perturbation vectors with an evaluated probability of success around 2^{-40} for the steps between 17 and 80. However, this probability can be affected by the NL-characteristic. Thus, we build the NL-characteristics corresponding to each matching perturbation vector in order to compute the exact probability of success. This aspect of the search will be further detailed in the next section. The perturbation vector we chose, which has 42 conditions between step 17 and 80, is given in Table 2.

4 Boomerang Attacks for SHA-0

Now that we found good perturbation vectors by relaxing certain conditions, a problem remains. Indeed, no message modification and no NL-characteristic are known for those vectors, and this makes the attack complexity drastically increase. This is the major drawback of Wang *et al.* collision attack on SHA-0 and other hash functions: is not easily reusable and we are stuck with their perturbation vector. Hopefully, some work have been done recently in order to theorize Wang *et al.* major improvements. Recently, De Cannière and Rechberger [5] introduced an automated tool that generates non linear part of a differential path, thus resolving the NL-characteristic problem. Then, Joux and Peyrin [14] provided a framework that generalizes message modifications and neutral bits. Thanks to the so-called *boomerang attack*, they describe techniques that allows an attacker to easily use neutral bits or message modifications, given only a main perturbation vector. In fact, boomerang attacks and NL-characteristic automated search are exactly the two tools we need for our attack to be feasible. Finally, we replace the loss of the message modifications because of the new vector by the gain of the boomerang attack, which is a much more practical technique and fit for our constraints.

Boomerang attacks for hash functions can be seen as a generalization of collision search speed up techniques such as neutral bits or message modification. However, new possibilities were also suggested. In the usual setting, the attacker first sets the differential path and then tries to find neutral bits or message modifications if possible. In the *explicit conditions approach* from boomerang attacks, the attacker first set some constraints on the registers and the message words and then tries to find a differential path taking in account those constraints. One can see that for the latter part the NL-characteristic automated search tool becomes really handy. The constraints are set in order to provide very good message modifications or neutral bits that would not exist with a random differential path, or with very low probability. More generally, this can be seen as an *auxiliary characteristic*, different from the main one, but only fit for a few steps and this auxiliary characteristic can later be used as a neutral bit or a message modification, with very high probability (generally probability equal to 1) thanks to the preset constraints. Obviously the complexity of the collision search will decrease by adding as much auxiliary characteristics as possible.

Building an auxiliary path requires the same technique as for a main path, that is the local collisions. We refer to [14] for more details on this construction for `SHA-1`, since the technique is identically applicable to `SHA-0`. In our attack, we will consider two different types of auxiliary paths and we will use them as neutral bits (and not message modifications). Informally, we define the range of an auxiliary path to be the latest step where the uncontrolled differences from the auxiliary path (after the early steps) do not interfere in the main differential path. The first one, AP_1, will have very few constraints but the range will be low. On contrary, the second type, AP_2, will require a lot of constraints but the range will be much bigger. A trade-off among the two types needs to be considered in order not to have to many constraints forced (which would latter makes the NL-characteristic automated search tool fail) but also have a good set of auxiliary differential paths. More precisely, AP_1 and AP_2 are detailed in Figures 3 and 4 respectively. AP_1 is build upon only one local collision but the first uncontrolled difference appears at step 20. AP_2 is build upon three local collisions but the first uncontrolled difference appears at step 25. Note that, as remarked in the original paper from Joux

	W_0 to W_{15}	W_{16} to W_{31}
perturbation mask	0000001000000000	
differences on W^j	0000001000000000	0000101101100111
differences on W^{j+5}	0000000100000000	0000010110110011
differences on W^{j-2}	0000000000010000	000$\underline{1}$001000000010

i	A_i	W_i
-1:	--------------------------------	
00:	--------------------------------	--------------------------------
01:	--------------------------------	--------------------------------
02:	--------------------------------	--------------------------------
03:	--------------------------------	--------------------------------
04:	--------------------------------	--------------------------------
05:	---------------------------b----	--------------------------------
06:	---------------------------b----	-----------------------------a--
07:	-----------------------------a--	------------------------ā-------
08:	-------------------------------0	--------------------------------
09:	-------------------------------1	--------------------------------
10:	--------------------------------	--------------------------------
11:	--------------------------------	-------------------------------ā
12:	--------------------------------	--------------------------------
13:	--------------------------------	--------------------------------
14:	--------------------------------	--------------------------------
15:	--------------------------------	--------------------------------

Fig. 3. Auxiliary differential path AP_1 used during the attack. The first table shows the 32 first steps of the perturbation vector (with the first uncontrolled difference on registers at step 20) and the second gives the constraints forced in order to have probability one local collisions in the early steps in the case where the auxiliary path is positioned at bit $j = 2$. The MSB's are on the right and "-" stands for no constraint. The letters represent a bit value and its complement is denoted by an upper bar on the corresponding letter (see [14] for the notations).

and Peyrin, an auxiliary differential path used as a neutral bit with the first uncontrolled difference at step s is a valid neutral bit for step $s+3$ with a very high probability (the uncontrolled difference must first propagate before disrupting the main differential path). Thus, in our attack, we will use AP_1 and AP_2 as neutral bits for steps 23 and 28 respectively; that is as soon as we will find a conformant message pair up to those step during the collision search, we will trigger the corresponding auxiliary path in order to duplicate the conformant message pair. This will directly provide new conformant message pairs for free.

The next step is now to build a main differential path with the tool from De Cannière and Rechberger, containing as much auxiliary paths as possible (of course while favoring AP_2 instead of AP_1, the latter being less powerful). We refer to [5] for the details of this algorithm. The tool works well for SHA-0 as for SHA-1 and given a random chaining variable, it is easy to find a main differential path containing at least five auxiliary paths, with at least three AP_2 characteristics.

	W_0 to W_{15}	W_{16} to W_{31}
perturbation mask	1010000000100000	
differences on W^j	1010000000100000	000000001̲0110110
differences on W^{j+5}	0101000000010000	0000000001011011
differences on W^{j-2}	0001111100000011	0000000000001110

i	A_i	W_i
-1:	---------------------------d----	
00:	---------------------------d----	-----------------------------a--
01:	---------------------------e-a--	------------------------a̅-------
02:	---------------------------e---1	-----------------------------b--
03:	-----------------------------b-0	------------------------b̅------a̅
04:	-------------------------------0	-------------------------------a̅
05:	-------------------------------0	-------------------------------a̅
06:	--------------------------------	-------------------------------b̅
07:	--------------------------------	-------------------------------b̅
08:	--------------------------------	--------------------------------
09:	---------------------------f----	--------------------------------
10:	---------------------------f----	-----------------------------c--
11:	-----------------------------c--	------------------------c̅-------
12:	-------------------------------0	--------------------------------
13:	-------------------------------0	--------------------------------
14:	--------------------------------	-------------------------------c̅
15:	--------------------------------	-------------------------------c̅

Fig. 4. Auxiliary differential path AP_2 used during the attack. The first table shows the 32 first steps of the perturbation vector (with the first uncontrolled difference on registers at step 25) and the second gives the constraints forced in order to have probability one local collisions in the early steps in the case where the auxiliary path is positioned at bit $j = 2$. The MSB's are on the right and "-" stands for no constraint. The letters represent a bit value and its complement is denoted by an upper bar on the corresponding letter (see [14] for the notations).

Note that this part, as the automated tool, is purely heuristic and often more auxiliary paths can be forced[2]. However, the behavior of the automated tool is quite dependant of the perturbation vector. Thus, among the possible good ones, we chose a perturbation vector (depicted in Table 2) that seemed to work well with the automated search tool. Note that since the perturbation vector remains the same during the two parts of the attack, this property will be true for the two blocks of message.

[2] The auxiliary path AP_2 has one condition on the IV (the bit d in Figure 4) and this harden the task of the attacker to place a lot of them in the main differential trail. However, this is already taken in account in the evaluation. There remains space for improvements on this part but we chose to describe an easy-to-implement attack instead of the best possible but hard to implement one.

5 The Final Collision Attack

At this point, we have all the required elements to mount the attack and analyze it. Its total complexity will be the addition of the complexities of the collision search for both blocks[3]. Note that, unlike for the 2-block collision attacks for SHA-1 where the first block complexity is negligible compared to the second block one, here our perturbation vector imposes the same raw complexity for both blocks.

5.1 A Method of Comparison

As observed in [6,14], there are many different collision search speeding techniques for the SHA family of hash functions. However, their real cost is often blurry and it becomes hard to compare them. Thus, it has been advised to measure the efficiency of those tools with an efficient implementation of the various hash functions attacked, for example by using OpenSSL [22]. For any computer utilized, one can give the complexity of the attack in terms of number of function calls of the hash function with an efficient implementation, which is relatively independent of the computer used.

In their paper [18], Naito *et al.* claimed a complexity of 2^{36} SHA-0 calls for their collision attack. However, their implementation required approximatively 100 hours on average in order to find one collision on a PentiumIV 3,4 GHz. 100 hours of SHA-0 computations on this processor would correspond to $2^{40,3}$ SHA-0 calls approximatively with OpenSSL, which is far from the claimed complexity.

Thus, in this paper, we chose to handle the complexities in terms of number of SHA-0 calls with OpenSSL in order to allow an easy comparison. The time measurements have been done on a single PC with an AMD Opteron 2, 2 GHz processor.

5.2 Without Collision Search Speedup

Without even using boomerang attacks, our new differential paths already provides an improvement on the best known collision attack against SHA-0. Indeed, in our perturbation vector, 42 conditions remain after step 17. However, by refining the differential path utilized (i.e. by forcing some conditions just before step 17, without any impact on the complexity since located in the early steps), one can easily take care of the two conditions from step 17 before beginning the collision search. Therefore, we are finally left with 40 conditions per message block. Since for each basic message pair tested during the collision search only one quarter of a whole SHA-0 is computed in average, we expect a complexity of $2^{40}/2^{2} = 2^{38}$ SHA-0 evaluations for one block and thus a total complexity of 2^{39} SHA-0 evaluations for one complete collision. This theoretical complexity is fully confirmed by practical implementation ($2^{37,9}$ and $2^{37,8}$ SHA-0 evaluations

[3] One can argue that the cost of the NL-characteristics automated search tool has also to be counted. However, unlike in the SHA-1 case, for SHA-0 the number of possible collisions that can be generated with only one full differential path construction is really big. Thus, this cost becomes largely negligible compared to the collision search.

Table 3. Message instance for a 2-block collision: $H(M_1, M_2) = H(M_1', M_2') = A_2||B_2||C_2||D_2||E_2$, computed according to the differential path given in Tables 5, 6, 7 and 8 from appendix

	1^{st} block		2^{nd} block	
	M_1	M_1'	M_2	M_2'
W_0	0x4643450b	0x46434549	0x9a74cf70	0x9a74cf32
W_1	0x41d35081	0x41d350c1	0x04f9957d	0x04f9953d
W_2	0xfe16dd9b	0xfe16dddb	0xee26223d	0xee26227d
W_3	0x3ba36244	0x3ba36204	0x9a06e4b5	0x9a06e4f5
W_4	0xe6424055	0x66424017	0xb8408af6	0x38408ab4
W_5	0x16ca44a0	0x96ca44a0	0xb8608612	0x38608612
W_6	0x20f62444	0xa0f62404	0x8b7e0fea	0x0b7e0faa
W_7	0x10f7465a	0x10f7465a	0xe17e363c	0xe17e363c
W_8	0x5a711887	0x5a7118c5	0xa2f1b8e5	0xa2f1b8a7
W_9	0x51479678	0xd147963a	0xca079936	0x4a079974
W_{10}	0x726a0718	0x726a0718	0x02f2a7cb	0x02f2a7cb
W_{11}	0x703f5bfb	0x703f5bb9	0xf724e838	0xf724e87a
W_{12}	0xb7d61841	0xb7d61801	0x37ffc03a	0x37ffc07a
W_{13}	0xa5280003	0xa5280041	0x53aa8c43	0x53aa8c01
W_{14}	0x6b08d26e	0x6b08d26c	0x90811819	0x9081181b
W_{15}	0x2e4df0d8	0xae4df0d8	0x312d423e	0xb12d423e

A_2	B_2	C_2	D_2	E_2
0x6f84b892	0x1f9f2aae	0x0dbab75c	0x0afe56f5	0xa7974c90

on average for the first and second blocks respectively). Our computer needs 39 hours on average in order to find a collision, which is much faster than Naito *et al.*'s attack and this with a less powerful processor.

5.3 Using the Boomerang Improvement

As one can use many collision search speedup techniques, a good choice can be the boomerang attack for its simplicity of use. We give in the appendix a possible differential path for the first block (Tables 5 and 6) and the second block (Tables 7 and 8). The notations used are also given in the appendix in Table 4. The chaining variable of the second block is the output of the first valid message pair found (i.e. conformant to the whole differential path) for the first block. As for the previous subsection, we are left with 40 conditions for each blocks since the two conditions from step 17 can be easily cancelled before the collision search. The differential path for the second block possesses 5 auxiliary paths. However, as explained before, it is possible to build NL-characteristics containing more auxiliary paths but we only take in account the average case and not the best case of behavior of the automated NL-characteristics search tool. For example, one can check that there are 3 AP_2 auxiliary paths in bit positions $j = \{17, 22, 30\}$ and 2 AP_1 auxiliary paths in bit positions $j = \{9, 11\}$ for the second block. The first block case is particular since the chaining variable is the predefined IV which is highly structured. This particular structure greatly improves our capability to place auxiliary paths since the condition on the chaining variable for AP_2 is verified on much more bit positions than what someone would expect in the random case. Thus, for the first block one can place 2 more AP_2 on average and one can check in Table 5 that there are 5 AP_2 auxiliary paths in bit positions $j = \{10, 14, 19, 22, 27\}$ and 2 AP_1 auxiliary paths in bit positions $j = \{9, 11\}$.

In theory, with k auxiliary paths, one would expect an improvement of a factor 2^k. However, this slightly depends on the type and the number of auxiliary paths used. Obviously, compared to the AP_1 auxiliary path, using the AP_2 type is better during the collision search. Thus, we expect an improvement of the attack of a factor approximatively $2^7 = 128$ for the first block and $2^5 = 32$ for the second one. We get something close in practice with a measured complexity of $2^{32,2}$ and 2^{33} function calls for the first and second block respectively. This leads to a final complexity for a collision on the whole SHA-0 of $2^{33,6}$ function calls, which compares favourably in theory and practice to the best known collision attack on this hash function: our computer can generate a 2-block collision for SHA-0 in approximatively one hour on average (instead of 100 hours of computation on a faster processor for the best known attack). For proof of concept, we provide in Table 3 a 2-block message pairs that collides with SHA-0.

6 Conclusion

In this paper, we introduced a new attack against SHA-0. By relaxing the previously established constraints on the perturbation vector, we managed to find better candidates. Then, as a collision search speedup technique, we applied on those candidates the boomerang attack which provides a good improvement with a real practicality of use. This work leads to the best collision attack on SHA-0 so far, requiring only one hour of computation on an average PC. Yet, there stills space for further improvements as some parts of the attack are heuristic. Moreover, this work shows the efficiency of the dual use of the boomerang attack for hash functions combined with a differential path automated search tool.

References

1. Bellare, M., Ristenpart, T.: Multi-Property-Preserving Hash Domain Extension and the EMD Transform. In: Lai, X., Chen, K. (eds.) ASIACRYPT 2006. LNCS, vol. 4284, pp. 299–314. Springer, Heidelberg (2006)
2. Biham, E., Chen, R.: Near-Collisions of SHA-0. In: Franklin, M.K. (ed.) CRYPTO 2004. LNCS, vol. 3152, pp. 290–305. Springer, Heidelberg (2004)
3. Biham, E., Chen, R., Joux, A., Carribault, P., Lemuet, C., Jalby, W.: Collisions of SHA-0 and Reduced SHA-1. In: Cramer, R. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 36–57. Springer, Heidelberg (2005)

4. Biham, E., Dunkelman, O.: A Framework for Iterative Hash Functions: HAIFA. In: Proceedings of Second NIST Cryptographic Hash Workshop (2006), www.csrc.nist.gov/pki/HashWorkshop/2006/program_2006.htm
5. De Cannière, C., Rechberger, C.: Finding SHA-1 Characteristics: General Results and Applications. In: Lai, X., Chen, K. (eds.) ASIACRYPT 2006. LNCS, vol. 4284, pp. 1–20. Springer, Heidelberg (2006)
6. De Cannire, C., Mendel, F., Rechberger, C.: Collisions for 70-step SHA-1: On the Full Cost of Collision Search. In: Adams, C., Miri, A., Wiener, M. (eds.) SAC 2007. LNCS, vol. 4876. Springer, Heidelberg (2007)
7. Chabaud, F., Joux, A.: Differential Collisions in SHA-0. In: Krawczyk, H. (ed.) CRYPTO 1998. LNCS, vol. 1462, pp. 56–71. Springer, Heidelberg (1998)
8. Damgård, I.: A Design Principle for Hash Functions. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 416–427. Springer, Heidelberg (1990)
9. Dean, R.D.: Formal Aspects of Mobile Code Security. PhD thesis, Princeton University (1999)
10. Dobbertin, H.: Cryptanalysis of MD4. In: Gollmann, D. (ed.) FSE 1996. LNCS, vol. 1039, pp. 53–69. Springer, Heidelberg (1996)
11. Menezes, A.J., Vanstone, S.A., Van Oorschot, P.C.: Handbook of Applied Cryptography. CRC Press, Inc., Boca Raton (1996)
12. Hoch, J.J., Shamir, A.: Breaking the ICE - Finding Multicollisions in Iterated Concatenated and Expanded (ICE) Hash Functions. In: Robshaw, M.J.B. (ed.) FSE 2006. LNCS, vol. 4047, pp. 179–194. Springer, Heidelberg (2006)
13. Joux, A.: Multi-collisions in Iterated Hash Functions. In: Franklin, M. (ed.) CRYPTO 2004. LNCS, vol. 3152, pp. 306–316. Springer, Heidelberg (2004)
14. Joux, A., Peyrin, T.: Hash Functions and the (Amplified) Boomerang Attack. In: Menezes, A. (ed.) CRYPTO 2007. LNCS, vol. 4622, pp. 244–263. Springer, Heidelberg (2007)
15. Kelsey, J., Schneier, B.: Second Preimages on n-bit Hash Functions for Much Less Than 2^n Work. In: Cramer, R. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 474–490. Springer, Heidelberg (2005)
16. Manuel, S.: Cryptanalyses Différentielles de SHA-0. Mémoire pour l'obtention du Mastère Recherche Mathematiques Applications au Codage et à la Cryptographie. Université Paris 8 (2006), http://www-rocq.inria.fr/codes/Stephane.Manuel
17. Merkle, R.C.: One Way Hash Functions and DES. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 428–446. Springer, Heidelberg (1990)
18. Naito, Y., Sasaki, Y., Shimoyama, T., Yajima, J., Kunihiro, N., Ohta, K. (eds.): ASIACRYPT 2006. LNCS, vol. 4284, pp. 21–36. Springer, Heidelberg (2006)
19. National Institute of Standards and Technology. FIPS 180: Secure Hash Standard (May 1993), http://csrc.nist.gov
20. National Institute of Standards and Technology. FIPS 180-1: Secure Hash Standard (April 1995), http://csrc.nist.gov
21. National Institute of Standards and Technology. FIPS 180-2: Secure Hash Standard (August 2002), http://csrc.nist.gov
22. OpenSSL. The Open Source toolkit for SSL/TLS (2007), http://www.openssl.org/source
23. Rivest, R.L.: RFC 1321: The MD5 Message-Digest Algorithm (April 1992), http://www.ietf.org/rfc/rfc1321.txt
24. Rivest, R.L.: RFC 1320: The MD4 Message Digest Algorithm (April 1992), http://www.ietf.org/rfc/rfc1320.txt
25. Wagner, D.: The Boomerang Attack. In: Knudsen, L.R. (ed.) FSE 1999. LNCS, vol. 1636, pp. 156–170. Springer, Heidelberg (1999)

26. Wang, X., Lai, X., Feng, D., Chen, H., Yu, X.: Cryptanalysis of the Hash Functions MD4 and RIPEMD. In: Cramer, R. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 1–18. Springer, Heidelberg (2005)
27. Wang, X., Yu, H.: How to Break MD5 and Other Hash Functions. In: Cramer, R. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 19–35. Springer, Heidelberg (2005)
28. Wang, X., Yu, H., Yin, Y.L.: Efficient Collision Search Attacks on SHA-0. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 1–16. Springer, Heidelberg (2005)
29. Wang, X., Yin, Y.L., Yu, H.: Finding Collisions in the Full SHA-1. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 17–36. Springer, Heidelberg (2005)

Appendix

Table 4. Notations used in [5] for a differential path: x represents a bit of the first message and x^* stands for the same bit of the second message

(x, x^*)	$(0,0)$	$(1,0)$	$(0,1)$	$(1,1)$
?	✓	✓	✓	✓
-	✓	-	-	✓
x	-	✓	✓	-
0	✓	-	-	-
u	-	✓	-	-
n	-	-	✓	-
1	-	-	-	✓
#	-	-	-	-

(x, x^*)	$(0,0)$	$(1,0)$	$(0,1)$	$(1,1)$
3	✓	✓	-	-
5	✓	-	✓	-
7	✓	✓	✓	-
A	-	✓	-	✓
B	✓	✓	-	✓
C	-	-	✓	✓
D	✓	-	✓	✓
E	-	✓	✓	✓

Table 5. Steps 1 to 39 of the main differential path of the first block

i	A_i	W_i
-4:	00001111010010111000011111000011	
-3:	01000000110010010101000111011000	
-2:	01100010111010110111001111111010	
-1:	11101111110011011010101110001001	
00:	01100111010001010010001100000001	0100111001001011000001010n0010u1
01:	111011011111111110011101 1n1111u0	0100000011011011010100001n000000
02:	01100111011111111101n10101101010	1111011000011110100111011n011011
03:	00110101001010 0n00001100n0uuuuu0	0011100010101001011100100u000101
04:	111100000nu000001010110001110000	u1100100010000000010100000u0101n1
05:	00111n0000000100110001000000u0n1u1	n0010100110010000101010010100000
06:	1011010111011011000010 1u100u1001	n0100010111101000011110 00u000100
07:	100unnnnnnnnn0100nu0100101u11001	00010010111110100010101100101101 0
08:	1000011100001n000n100u0n010nn001	0101101001110001000110001n0001u1
09:	0010000000000010un00nu1u1un01100	n1010001010001111001011 00u1110n0
10:	1110011010010 1000nu01u10un00n100	01111010011000100100011100011000
11:	01111000111000110 1nuu10101000101	0111000100110111010110011u1110u0
12:	0100110101101000001 0u0000n110000	10110111110101-----1-----u000001
13:	010110011100000----010-0-01001u0	101001010----------------n0000u1
14:	10111100--------------1--110u011	01101-0----0--1----0---0-1-011u0
15:	10100------------------0-1-u0100	n0101-0----0--1----0---0-1-11000
16:	--01-----------------------n0011	010001110----------------00101n0
17:	-----------------------------1n-	n1000-0----1--1----1---0-u-10011
18:	1----------------------------0--	01000-0----1--1----0---0-0-011u0
19:	--------------------------------	n00110100----------------0001011
20:	--------------------------------	n0110-0----1-------0-----0-000u1
21:	------------------------------n-	u1100-1------------------u-10111
22:	--------------------------------	00001-1------------------0-00110
23:	------------------------------n-	n1011-1----0-------0-----u-11001
24:	--------------------------------	u0000-0------------------1-11100
25:	------------------------------n-	01101-1------------------u-10111
26:	--------------------------------	u1010-1----0-------1-----0-011u0
27:	--------------------------------	01001-1------------------0-01110
28:	--------------------------------	u0000-0------------------1-11011
29:	--------------------------------	u0111-0------------------0-00010
30:	--------------------------------	01101-1------------------1-10010
31:	--------------------------------	10110-1------------------0-01001
32:	--------------------------------	00111-1------------------1-00100
33:	--------------------------------	01011-1------------------1-11101
34:	--------------------------------	00010-0------------------0-010u0
35:	------------------------------u-	10001-0------------------n-10110
36:	--------------------------------	11100-0------------------0-000u1
37:	--------------------------------	n0010-0------------------0-001u0
38:	------------------------------u-	n1101-0------------------n-11110
39:	--------------------------------	n1100-1------------------0-001n0
	...	...

Table 6. Steps 40 to 80 of the main differential path of the first block

i	A_i	W_i
	...	...
40:	--------------------------------	n1111-0------------------0-10000
41:	--------------------------------	n0010-1------------------0-11010
42:	--------------------------------	n0100-0------------------1-110u1
43:	------------------------------u-	00000-1------------------n-01010
44:	--------------------------------	00011-0------------------0-100n0
45:	--------------------------------	n0111-1------------------1-10110
46:	--------------------------------	n0111-1------------------0-00010
47:	--------------------------------	u0010-1------------------1-00000
48:	--------------------------------	01101-0------------------0-010n0
49:	------------------------------n-	11111-1------------------u-10011
50:	--------------------------------	01000-1------------------0-100u0
51:	--------------------------------	u1110-1------------------0-10010
52:	--------------------------------	n1101-1------------------1-11110
53:	--------------------------------	n0001-1------------------1-001u0
54:	------------------------------u-	11011-0------------------n-11110
55:	--------------------------------	10001-0------------------0-000n0
56:	--------------------------------	n0111-1------------------0-001n1
57:	------------------------------n-	n0110-1------------------u-11101
58:	--------------------------------	u1110-1------------------1-11001
59:	------------------------------n-	u1110-0------------------u-010u1
60:	------------------------------u-	n1111-1------------------n-100n1
61:	--------------------------------	01010-0------------------0-010n1
62:	--------------------------------	01111-1------------------1-11111
63:	--------------------------------	10011-1------------------0-00010
64:	--------------------------------	n1000-0------------------0-10110
65:	--------------------------------	01000-0------------------1-00011
66:	--------------------------------	01000-0------------------0-101u1
67:	------------------------------u-	01001-0------------------n-01001
68:	--------------------------------	10001-0------------------0-100u0
69:	--------------------------------	u0010-1------------------1-11000
70:	--------------------------------	u1010-0------------------1-011n1
71:	------------------------------n-	u0101-0------------------u-01101
72:	--------------------------------	00011-1------------------0-100u0
73:	--------------------------------	n1010-1------------------0-11000
74:	--------------------------------	n1100-0------------------0-10010
75:	--------------------------------	u1110-1------------------1-110n1
76:	------------------------------n-	11011-1------------------u-00100
77:	--------------------------------	00111-0------------------1-000n1
78:	--------------------------------	n0011-0------------------1-11101
79:	--------------------------------	u0101-0------------------1-01000
80:	--------------------------------	

Table 7. Steps 1 to 39 of the main differential path of the second block

i	A_i	W_i
-4:	111011010000101010001110101010u1	
-3:	01000110011100010110101100101000	
-2:	10010000011011111010001110100111	
-1:	11100110001011011100010100001001	
00:	01011110101010111100001100111101	1001101001110110110011110u1100n0
01:	u0111011010011100010111unn1010n1	0000010010111001100101010u111101
02:	11010011001110011011u000110u0111	1110111000100100001000100n111101
03:	111001111000010u000unnnnnn000100	1001101001000110011001001n110101
04:	u0100101unn01010000u100011110110	u0111000010000000000010101u1101u0
05:	n000un001000011u00100000000nn0n0	u0111000011000000000011000010010
06:	nnn00100010111100111 00n1nu1u011u	u0001011011111101000010 11u101010
07:	10nuuuuuuuuuuuuu11100n00un0u1001	11100001011111111111011000111100
08:	0001111100000000unnn11010001n001	1010001011110001101110001u1001n1
09:	00000111111111111110001n111un111	u1001010000001111001100 10n1101u0
10:	1110110110111111110100nu111uu011	00000010111100001010011111001011
11:	0011111001000101001101 1uu0n000u0	1111011101100100111010101n1110n0
12:	0100011010001110001111 11nuu1u011	001101111111-------------n111010
13:	101010000000----0--------01111u0	01010--------------------u0000u1
14:	00110001-----------------1010010	10010------0----1--------00110n1
15:	10011--------------------10101n0	n0110------0----1--------0111110
16:	0-------------------------011000	10000--------------------11010u1
17:	1-----------------------------n-	u1000------1----1--------u100111
18:	0-------------------------------	01100------1----0--------01111u0
19:	--------------------------------	n1010--------------------1110100
20:	--------------------------------	u1000------1-------------10000n1
21:	------------------------------n-	n1101--------------------u010011
22:	--------------------------------	11101--------------------1100010
23:	------------------------------n-	u1011------0-------------u110101
24:	--------------------------------	n1001--------------------0010110
25:	------------------------------u-	00010--------------------n001011
26:	--------------------------------	u0001------1-------------01110u0
27:	--------------------------------	10111--------------------0011001
28:	--------------------------------	n1110--------------------1101001
29:	--------------------------------	u0000--------------------0010100
30:	--------------------------------	01000--------------------0001001
31:	--------------------------------	01011--------------------1000101
32:	--------------------------------	00101--------------------1010111
33:	--------------------------------	11000--------------------0010001
34:	--------------------------------	01110--------------------00000n0
35:	------------------------------n-	10101--------------------u101001
36:	--------------------------------	10011--------------------10110u1
37:	--------------------------------	n1000--------------------01100u0
38:	------------------------------u-	n1001--------------------n010100
39:	--------------------------------	n0001--------------------11000n0
	...	...

Table 8. Steps 40 to 80 of the main differential path of the second block

i	A_i	W_i
	. . .	. . .
40:	--------------------------------	u0101--------------------1001001
41:	--------------------------------	n0100--------------------0010111
42:	--------------------------------	u0000--------------------01100u1
43:	------------------------------u-	00111--------------------n101101
44:	--------------------------------	10001--------------------01011n0
45:	--------------------------------	n0011--------------------1010000
46:	--------------------------------	n0011--------------------1100111
47:	--------------------------------	n0011--------------------0011000
48:	--------------------------------	11101--------------------10011u0
49:	------------------------------u-	01010--------------------n001000
50:	--------------------------------	01110--------------------11100n0
51:	--------------------------------	n0111--------------------0111000
52:	--------------------------------	n0001--------------------1101011
53:	--------------------------------	n0100--------------------11100u0
54:	------------------------------u-	11000--------------------n000010
55:	--------------------------------	00111--------------------00001n0
56:	--------------------------------	u1100--------------------10001u0
57:	------------------------------u-	u0001--------------------n110000
58:	--------------------------------	n1000--------------------1101011
59:	------------------------------u-	u1111--------------------n0000u1
60:	------------------------------u-	n0010--------------------n0100n0
61:	--------------------------------	01100--------------------10100n1
62:	--------------------------------	11001--------------------0101000
63:	--------------------------------	01100--------------------0000100
64:	--------------------------------	n0011--------------------0101001
65:	--------------------------------	00101--------------------0101000
66:	--------------------------------	01011--------------------11101n0
67:	------------------------------n-	11111--------------------u100000
68:	--------------------------------	11110--------------------10100n1
69:	--------------------------------	n0100--------------------1010011
70:	--------------------------------	n0010--------------------00011n0
71:	------------------------------n-	n0100--------------------u100001
72:	--------------------------------	10011--------------------10101u1
73:	--------------------------------	n1001--------------------0010111
74:	--------------------------------	n0101--------------------1101110
75:	--------------------------------	u1111--------------------11001n1
76:	------------------------------n-	01100--------------------u111110
77:	--------------------------------	00001--------------------11010n0
78:	--------------------------------	n0111--------------------1101000
79:	--------------------------------	n0001--------------------0110011
80:	--------------------------------	

The Hash Function Family LAKE

Jean-Philippe Aumasson[1,⋆], Willi Meier[1], and Raphael C.-W. Phan[2,⋆⋆]

[1] FHNW, 5210 Windisch, Switzerland
[2] Electronic & Electrical Engineering, Loughborough University, LE11 3TU,
United Kingdom

Abstract. This paper advocates a new hash function family based on the HAIFA framework, inheriting built-in randomized hashing and higher security guarantees than the Merkle-Damgård construction against generic attacks. The family has as its special design features: a nested feedforward mechanism and an internal wide-pipe construction within the compression function. As examples, we give two proposed instances that compute 256- and 512-bit digests, with a 8- and 10-round compression function respectively.

Keywords: Hash function, HAIFA, Randomized hashing, Salt, Wide-pipe.

1 Introduction

Why do we need another hash function? Aside from the explicit aim of the U.S. Institute of Standards and Technology (NIST) to revise its standards [30, 29], motivations lie in the present status of hash functions: among the proposals of recent years, many have been broken (including "proven secure" ones), or show impractical parameters and/or performance compared to the SHA-2 functions, despite containing interesting design ideas[1]. For example all the hash functions proposed at FSE in the last five years [18,17,19] are now broken [35,25,34], except for one [33] based on SHA-256. We even see recent works [27] breaking old designs that had until now been assumed secure due to absence of attacks. It seems necessary to learn from these attacks and propose new hash functions that are more resistant to known cryptanalysis methods, especially against differential-based attacks, which lie at the heart of major hash function breaks. These design proposals would also hopefully contribute to the discovery of new ways to attack hash functions, contributing to NIST's SHA3 development effort.

This paper introduces the hash function family LAKE along with two particular instances aimed at suiting a wide variety of cryptographic usages as well as present and future API's. We adopted the extended Merkle-Damgård framework HAIFA [8] and include the following features in our design:

[⋆] Supported by the Swiss National Science Foundation under project number 113329.

[⋆⋆] Work done while the author was with the Security & Cryptography Lab (LASEC), Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland.

[1] Note that the ISO/IEC standards Whirlpool [3] and RIPEMD-160 [13] are not broken yet.

K. Nyberg (Ed.): FSE 2008, LNCS 5086, pp. 36–53, 2008.

- **Built-in salted hashing:** to avoid extra code for applications requiring a salt (be it random, a nonce, etc.), and to encourage the use of randomized hashing.
- **Software-oriented:** we target efficiency in software, although efficient hardware implementations are not precluded.
- **Direct security**: the security is not conditioned on any external hardness assumption.
- **High speed:** with significantly better performance than the SHA-2 functions on all our machines.
- **Flexibility:** with variable number of rounds and digest length.

Road Map. First we give a complete specification of LAKE (§2), then we explain our design choices (§3), present performance (§4), and study security of the proposed instances (§5). Appendices include lists of constants, and test values.

2 Specification

This section gives a bottom-up specification of the LAKE family, and the definition of the instances LAKE-256 and LAKE-512 ("LAKE" designates both the general structure and the family of hash functions built upon it, while instances have parametrized names). We'll meet the following symbols throughout the paper (length unit is the bit).

w	Length of a word	$\tilde{M}$	Padded message
n	Length of the chaining variable	$\tilde{M}^t$	t-th (padded) message block
m	Length of the message block	$\tilde{M}^t_i$	i-th word of $\tilde{M}^t$
s	Length of the salt	N	Number of blocks of $\tilde{M}$
d	Length of the digest	S	Salt
b	Length of the block index	S_i	i-th word of the salt
r	Number of rounds	H^t	t-th chaining variable
t	Index of the current block	H^t_i	i-th word of H^t
t_i	i-th word of t	D	Message digest
M	Message to hash	IV	n-bit fixed initial value

We let lengths n, m, etc. be multiples of w. Hexadecimal numbers are written in `typewriter` style, and function or primitive labels in sans-serif font. The operator symbols $+$, $\oplus$, $\gg$, $\ggg$, $\vee$, $\wedge$ keep their standard definition. Last but not least, LAKE is defined in *unsigned little-endian* representation.

2.1 Building Blocks

LAKE's compression function compress is made up of three procedures: *initialization* (function saltstate), internal *round function* (function processmessage), and *finalization* (function feedforward). Fig. 1 represents the interaction between these functions.

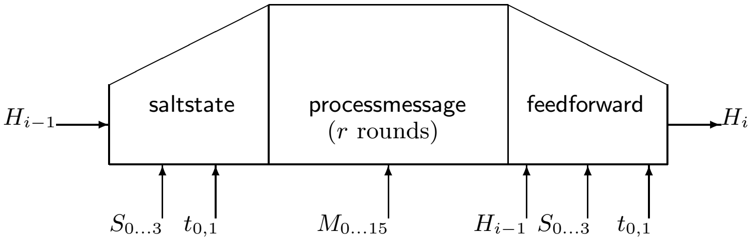


Fig. 1. The structure of LAKE's compression function: the chaining variable H_{i-1} is transformed into a local chaining variable twice as large, which undergoes message-dependent modification, before being shrunk to define H_i

The saltstate Function. This mixes the global chaining variable H with the salt S and the block index t, writing its output into the buffer L, twice as large as H. In addition, saltstate uses 16 constants $C_0, \ldots, C_{15}$, and a function g that maps a $4w$-bit string to a w-bit string. L plays the role of the chain variable—in a "wide-pipe" fashion—, while g can be seen as an internal compression function, and the constants are used to simulate different functions. In the following algorithm, word indexes are taken modulo the number of w-bit words in the array. For example, in the second "for" loop of saltstate's algorithm, i ranges from $2 \equiv 10 \pmod 8$ to $7 \equiv 15 \pmod 8$ for H_i, and over $2, 3, 0, 1, \ldots, 2, 3$ for S_i (see also Fig. 2).

```
saltstate
input $H = H_0 \| \ldots \| H_7$,   $S = S_0 \| \ldots \| S_3$,   $t = t_0 \| t_1$

1. for $i = 0, \ldots, 7$ do
       $L_i \leftarrow H_i$
2. $L_8 \leftarrow \mathsf{g}(H_0, S_0 \oplus t_0, C_8, 0)$
3. $L_9 \leftarrow \mathsf{g}(H_1, S_1 \oplus t_1, C_9, 0)$
4. for $i = 10, \ldots, 15$ do
       $L_i \leftarrow \mathsf{g}(H_i, S_i, C_i, 0)$

output $L = L_0 \| \ldots \| L_{15}$
```

The first eight words in the buffer L allow H to pass through saltstate unchanged for use in later feedforwarding, while the last eight words ensure dependence on

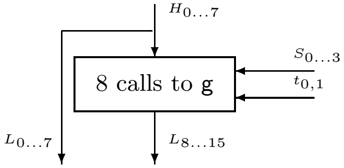


Fig. 2. The saltstate function

the salt and the block index. Clearly, the function is not surjective, but will be injective for a well-chosen g. These local properties do not raise any security issues (see §5 for more discussion).

The processmessage Function. This is the bulk of LAKE's round function. It incorporates the current message block M within the current internal chaining variable L, with respect to a permutation σ of the message words. It employs a local m-bit buffer F for local feedforward, and internal compression functions f and g, both mapping a $4w$-bit string to a w-bit string. In the algorithm below, indexes are reduced modulo 16, i.e. $L_{-1} = L_{15}$, $L_{16} = L_0$.

```
processmessage
input $L = L_0\|\ldots\|L_{15}$,   $M = M_0\|\ldots\|M_{15}$,   $\sigma$

1. $F \leftarrow L$
2. for $i = 0,\ldots,15$ do
       $L_i \leftarrow \mathsf{f}(L_{i-1}, L_i, M_{\sigma(i)}, C_i)$
3. for $i = 0,\ldots,15$ do
       $L_i \leftarrow \mathsf{g}(L_{i-1}, L_i, F_i, L_{i+1})$

output $L = L_0\|\ldots\|L_{15}$
```

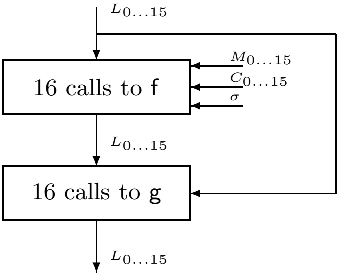


Fig. 3. The processmessage function

The feedforward Function. This compresses the initial global chaining variable H, the salt S, the block index t and the hitherto processed internal chaining variable L. It outputes the next chaining variable. In the algorithm indexes are reduced modulo 4 for S (see also Fig. 4).

```
feedforward
input $L = L_0\|\ldots\|L_{15}$,   $H = H_0\|\ldots\|H_7$,   $S = S_0\|\ldots\|S_3$,   $t = t_0\|t_1$

1. $H_0 \leftarrow \mathsf{f}(L_0, L_8, S_0 \oplus t_0, H_0)$
2. $H_1 \leftarrow \mathsf{f}(L_1, L_9, S_1 \oplus t_1, H_1)$
3. for $i = 2,\ldots,7$ do
       $H_i \leftarrow \mathsf{f}(L_i, L_{i+8}, S_i, H_i)$

output $H = H_0\|\ldots\|H_7$
```

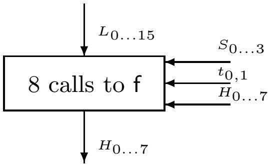


Fig. 4. The feedforward function

The compress Function. This is the *compression function* of LAKE. It computes the next chaining variable H^{t+1} from the current H^t, the current message block M^t, the salt S and the current block index t. The number of rounds r and the permutations $(\sigma_i)_{0\le i<r}$ are parameters to be set later.

```
compress
input $H = H_0\|\ldots\|H_7$,  $M = M_0\|\ldots\|M_{15}$,  $S = S_0\|\ldots\|S_3$,  $t = t_0\|t_1$

1. $L \leftarrow$saltstate$(H, S, t)$
2. for $i = 0, \ldots, r-1$ do
       $L \leftarrow$processmessage$(L, M, \sigma_i)$
3. $H \leftarrow$feedforward$(L, H, S, t)$

output $H$
```

Apart from its arguments, compress requires 32 memory words for L and F. Here, L acts as an internal chaining variable, twice as long as H, finally shrunk to output the next chaining variable, as in the "wide-pipe" construction [23,24]. The goal of this approach is to make local collisions difficult to find—if not impossible. Observe that the current message block M is input r times to (and thus input to r different points of) processmessage; meanwhile the salt S, the current block index t, and the chaining variable H are feedforwarded to the last stage of compress, thus in fact these inputs are injected into two different points of compress.

2.2 The LAKE Structure

The LAKE structure consists of the sequence: *initialization* (functions pad and init), *iteration* of compress, and *finalization* (function out). We start this section with a description of the padding rule, inherited from HAIFA.

Padding. A message M is padded by concatenating a '1' bit followed by sufficient number of '0' bits, then the b-bit message length, and the digest length d, such that a padded message $\tilde{M}$ is exactly km bits, for a minimal integer k.

Initialization. The effective initial chaining variable H^0 is computed by the function init on input an n-bit initial value IV and a length d of the output hash value:

$$H^0 \leftarrow \mathsf{init}(IV, d) = \mathsf{compress}(IV, d, 0, 0).$$

For words of reasonable length d is written in M_0, and all other M_i's are null. In practice H^0 should be precomputed, unless variable digest length is necessary.

Finalization. The function out simply truncates the final chaining variable H^N to its d first bits, to return the final hash output digest D.

Overall Hashing. A LAKE hash function take as input a message and a salt, and is parametrized by an initial value IV, a number of rounds r, a sequence of permutations $(\sigma_i)_{0\leq i<r}$, the word size w, and subsequently bit-lengths n, m, d, s, b.

```
LAKE
input $M = M^0\| \dots \|M^{N-1}$,   $S = S_0\| \dots \|S_3$

1. $\tilde{M} \leftarrow \mathsf{pad}(M)$
2. $H^0 \leftarrow \mathsf{init}(IV, d)$
3. for $t = 0, \dots, N-1$ do
       $H^{t+1} \leftarrow \mathsf{compress}(H^t, \tilde{M}^t, S, t)$
4. $D \leftarrow \mathsf{out}(H^N)$

output $D$
```

2.3 Instances

We introduce LAKE-256 and LAKE-512, respectively suited for 32- and 64-bit words:
LAKE-256 has parameters

$$\begin{array}{lll} n\ = 256 \text{ (chaining variable)} & d = 256 \text{ (digest)} & r = 8 \text{ (rounds)} \\ m = 512 \text{ (message block)} & s = 128 \text{ (salt)} & b = 64 \text{ (block index)} \end{array}$$

Its IV and constants $C_0, \dots, C_{15}$ are extracted from π (see Appendix A), and permutations of the set $\{0, \dots, 15\}$ are defined as in MD5 by

$$\begin{array}{llll} i \equiv 0 & (\text{mod } 4) \Rightarrow \sigma_i(j) = j & \\ i \equiv 1 & (\text{mod } 4) \Rightarrow \sigma_i(j) = 5j + 1 & (\text{mod } 16) \\ i \equiv 2 & (\text{mod } 4) \Rightarrow \sigma_i(j) = 3j + 5 & (\text{mod } 16) \\ i \equiv 3 & (\text{mod } 4) \Rightarrow \sigma_i(j) = 7j & (\text{mod } 16) \end{array}$$

The internal compression functions are[2]

$$\mathsf{f}(a, b, c, d) = \big[a + (b \vee C_0)\big] + \Big(\big[c + (a \wedge C_1)\big] \ggg 7\Big) + \Big(\big[b + (c \oplus d)\big] \ggg 13\Big)$$
$$\mathsf{g}(a, b, c, d) = \big[(a + b) \ggg 1\big] \oplus (c + d).$$

[2] It was observed by F. Mendel, C. Rechberger, and M. Schläffer [26] that the non-invertibility of f can be exploited to find collisions for a reduced version with 4 rounds (instead of 8) faster than with a birthday attack (this was independently suggested by S. Lucks at FSE 2008). These authors mention that choosing e.g. $\mathsf{f}(a, b, c, d) = [a + (b \vee C_0)] + ([d + (a \wedge C_1)] \ggg 7) + ([b + (c \oplus d)] \ggg 13)$ makes this attack impossible. It is however unclear whether their suggestion impacts the security of the original version with all 8 rounds.

LAKE-512 has parameters

$n = 512$ (chaining variable)	$d = 512$ (digest)	$r = 10$ (rounds)
$m = 1024$ (message block)	$s = 256$ (salt)	$b = 128$ (block index)

Constant values are given in Appendix A, permutations are the same as for LAKE-256, and internal compression functions are

$$\mathsf{f}(a,b,c,d) = \big[a + (b \vee C_0)\big] + \Big(\big[c + (a \wedge C_1)\big] \ggg 17\Big) + \Big(\big[b + (c \oplus d)\big] \ggg 23\Big)$$
$$\mathsf{g}(a,b,c,d) = \big[(a+b) \ggg 1\big] \oplus (c+d).$$

Other Instances. Instances with digest length $0 < d \leq 256$ take LAKE-256 parameters, and instances with $256 < d \leq 512$ take LAKE-512 parameters. Since the effective initial value H^0 depends on d, this will be distinct for each choice of d.

3 Design

Design rationale is given top-down, from the operating mode to the wordwise operators. Apart from the obvious concerns of security and speed guiding principles include:

- **Withstand differential attacks**: no high-probability differential path should be exploitable, including techniques based on impossible or truncated differentials.
- **Prevent side-channel leakage**: time and memory consumption as well as operations should be input-independent, to avoid weaknesses in keyed modes.
- **Facilitate implementation**: instances proposed should allow compact implementations, be the less processor-specific as possible, use simple operators.
- **Facilitate analysis**: we use a small number of building blocks, of reasonable complexity, and provide flexible instances, in a clear and concise specification.

3.1 HAIFA as the Operating Mode

Some properties of the classical MD mode and the need for salted hashing (not explicitly handled by the previous constructions) motivated the design of the HAsh Iterative FrAmework (HAIFA) [8]. Its main novelties are the explicit input of a salt and the number of bits hashed so far to the compression function, the computation of the initial value depending on the digest length, and the padding rule. Consequently,

- generic attacks for finding "one-of-many" preimages and second-preimages with k targets requires 2^d trials for HAIFA against $2^d/k$ for MD,
- online "herding" time-memory trade-off for finding preimages with memory 2^t require $2^{d/2+t/2+s}$ trials for HAIFA against $2^{d/2+t/2}$ for MD,

- HAIFA captures the MD, enveloped MD [5], RMC [1], ROX [2], and Wide-pipe [23,24] operating modes,
- it explicitly handles randomized hashing.

Other operating modes could have been chosen, however we believe that HAIFA offers more advantages, in addition to a simple and familiar framework for cryptanalysts and implementers since it builds on the well known MD mode.

On Salted Hashing. To the best of our knowledge, LAKE is the first concrete hash construction to include built-in salting. Although this is not a strict requirement for randomized hashing (see e.g. the RMX transforms of Halevi and Krawczyk [15]), it has the advantage of requiring no additional programming, so that the hash function can be directly used as a black box fed with a message and a salt. The main application of randomized hashing is digital signatures, enhancing security by reducing the security requirement of the underlying hash function from collision-resistance to second-preimage-like notions [15]. More generally, a salt may find applications in protocols requiring hash functions families, as well as future protocols may take advantage of it, be it public or secret, random or a counter. When a salt is not needed, the null value can be used.

A disadvantage of salted hashing is that extra-data might have to be sent, while disadvantages of built-in salt are that it might facilitate certain attacks, and potentially increases the complexity of the algorithm. On the other hand, it encourages the use of randomized hashing, and prevents from weak home-brewed construction.

3.2 Building Blocks

The structure of compress is similar in spirit to the one of the overall hash function: initialization, chained rounds, and finalization. Essential differences are that the round function here allows no collision for a fixed chaining variable, and that the same message is used at each iteration (up to a permutation of words). The goal of the internal wide-pipe is indeed to have an (almost) injective function built as a repetition of processmessage; the goal is to make local collisions unlikely for fixed chaining variable, salt, and counter. Also note that, contrary to the SHA functions, LAKE-256 does not admit easily found fixed-points.

For designing compress we create two levels of interdependence: fast diffusion can be seen as a propagation of *spatial interdependence* across words at any intermediate state within the hash function, and complicates attacks including those that exploit high-probability differentials. Analogously, the feedforward mechanism allows injected values to influence two different intermediate states at different times during the processing, which achieves *temporal interdependence.* Although wordwise diffusion as a spatial interdependence technique is widely known in cryptographic literature to increase resistance to common attacks, the notion of feedforwarding as a form of achieving temporal interdependence is less treated. In fact, this latter serves to complicate the perturb-and-correct strategy used in many hash function attacks that exploit inputs with well chosen differences. The central building block processmessage achieves both spatial and temporal interdependence by making use of multiple-blockwise chaining [4] and feedforwarding respectively. Spatial and temporal interdependencies are achieved in a similar way within feedforward and the structure of the compression function compress. We further comment on the three internal modules of compress hereafter.

We chose a round-dependent permutation for the message input, with same permutations as MD5; note that, though MD5 is broken, it is essentially due to the relative simplicity of its round function, rather than to the message input strategy. An alternative would be to use message expansion, as used in a fully XOR-linear fashion in SHA-0/1, and non-linearly in SHA-2. The main argument for recursive message expansion is that it simulates a complicated function, and the non-surjectivity makes collisions of random expanded messages useless. However, it may increase memory usage, and other strategies can be used to have a complex mixing.

3.3 Core Functions f and g

Each of these functions is called 136 times in an 8-round compress. We opted for a high-number of calls to simple functions, rather than a few calls to some complicated procedures, mainly because it simplifies analysis and implementation, and reduces the amount of code.

The role of f is to provide a large amount of mixing, to break linearity and diffuse changes across words. We considered various combinations of wordwise operators and our final choice was selected for its high ratio diffusion over speed, its ability to increase quickly the algebraic degree, and its simplicity. The much simpler g only aims at making each input influence the internal state, within a progressive diffusion of changes via addition carries and 1-bit rotations in a non-linear fashion. When used as arguments, constants C_i simulate distinct functions and reduces self-similarity.

3.4 Wordwise Operators

We chose a combination of standard constant-time word operators, known to be complementary to achieve cryptographic strength: integer addition and XOR diffuse changes locally, while logical operators AND and OR increase non-linearity–over GF(2) and GF(2^{32}). Though the operators AND and OR are in no way mandatory, the use of the sole triplet $(+, \oplus, \ggg)$ can be risky, as suggested by the existence of high-probability differentials in the stream ciphers Phelix and Salsa20/8, the block cipher TEA, or in the hash function FORK-256 [40,7,39,17]. Finally, rotation provides fast diffusion within the words, with a choice of data-independent distances—in order to avoid side-channel leakage, reduce the control of the attacker over the operations, and reduce complexity of the algorithm. Rotation counts of f were chosen so as to avoid byte alignments, and diffuse changes to any word offset as fast as possible; as observed by the authors of Twofish [37], one-bit rotation (as used in g) saves time over smartcards processors, compared to multi-bit rotation. We avoid integer multiplication essentially for performance reasons and the risk of timing leakage.

3.5 Parameters

We propose instances whose input and output have lengths similar to previous and current standards, to suit present and future API's, and minimize implementers' work. The salt length was chosen to be sufficient for randomized hashing, and to suit HAIFA's requirements (for which the salt should be at least half as large as the digest).

After intensive security analysis, we believe that eight rounds for LAKE-256 are sufficient for actual security, and as a security margin to counter future attacks (in comparison, MD5 and SHA-256 have four rounds, SHA-1 and SHA-512 have five rounds). We add two rounds for LAKE-512 whose larger state delays full diffusion.

4 Performance

4.1 Algorithmic Complexity

We consider here the algorithm independently from any specific implementation or parallelism issues, and study time and space complexities. However, from this abstract evaluation one cannot directly infer statements on the actual speed of the algorithms when implemented, which depends on a multitude of other factors (see speed benchmarks in §4.2).

Table 1 presents on its leftmost part the number of arithmetic operations for LAKE-256, LAKE-512, and their components, comparing with SHA-256 and SHA-512 (refering to [29]): our functions count slightly less operations than the SHA-2 equivalents, with significantly less XORs and more integer additions. The larger amount of rotations in SHA-2 functions increases the difference of operation counts on processors simulating a rotation with two shifts (as the Itanium and UltraSPARC). The rightmost part of Table 1 compares storage requirements.

Table 1. Algorithmic complexities and memory requirements (in bytes)

Function	Operations							Memory	
	Total	$+$	$\oplus$	$\ggg$	$\gg$	$\vee$	$\wedge$	ROM	RAM
f	10	5	1	2	0	1	1	-	-
g	4	2	1	1	0	0	0	-	-
saltstate	34	16	10	8	0	0	0	-	-
processmessage	224	112	32	48	0	16	16	-	-
feedforward	82	40	10	16	0	8	8	-	-
LAKE-256	1908	952	276	408	0	136	136	64	128
SHA-256	2232	600	640	576	96	0	320	256	64
LAKE-512	2356	1176	340	504	0	168	168	128	256
SHA-512	2632	712	752	672	96	0	400	512	128

4.2 Implementation

Implementation on 32- and 64-bit architectures should pose no problem, since we only use standard wordwise operators. On 8- and 16-bit architectures (e.g. smartcards) word operations have to be decomposed; rotations translate less simply than addition and XOR, but remain easily implementable. Choosing multiples of 8 as rotation counts would have improved performances on 8- and 16-bit processors, but reduced the quality of diffusion. We do not preclude hardware implementation, since our operators are consistently simple and fast.

Speed Benchmarks. It is, alas, rather difficult to make a complete and comprehensive relative study of hash functions: security evaluation requires intensive cryptanalysis effort (even for “proven secure” designs), and performances comparison cannot be fair when reference source codes are not published, or do not have a same degree of optimization, are more or less processor-dependent, etc., not to mention the issue of hardware benchmarks. The simplest solution is then to compare with the reference SHA-2 family, since other proposals would also be compared to these functions.

We compare LAKE-256 with SHA-256 using portable C implementations: respectively our reference code (available upon request) and the version in XySSL [12]. These codes have roughly the same level of optimization, and we used exactly the same source code for all processors. Cycles counts are measured using the RDTSC assembly instruction through the processor-specific cpucycles library [6], on machines running a Linux kernel 2.6.19 with Gentoo distributions; sources are compiled with `gcc 4.1.2` with full optimization flags (`-O3`) and processor-specific settings (e.g. `-march=pentium4`). For each machine, Table 2 shows the *median cycles count* measured among 1000 successive calls to the compression function with random inputs, along with the cycles-per-byte cost. This measurement has a relatively high variant, so we give rounded values for clarity, that seems sufficient for a raw comparison of performance.

LAKE-256 significantly outperforms SHA-256 on our test machines, particularly on our Athlon XP and Pentium D, while the difference of cycles on the other machines roughly matches the difference of arithmetic operations (see §4.1); this suggests that LAKE-256 takes a particular advantage of some feature of the former processors (possibly thread-level parallelism). These results

Table 2. Cycle counts for the compression function (and corresponding cycles-per-byte cost)

	CPU		Function	
Name	Frequency	L2 cache	LAKE-256	SHA-256
Athlon	800 MHz	256 Kb	2700 (42)	3000 (50)
Athlon XP	1830 MHz	512 Kb	2400 (38)	4500 (70)
Pentium 4	1500 MHz	256 Kb	3600 (56)	4050 (63)
Pentium 4	2400 MHz	512 Kb	3300 (52)	3900 (61)
Pentium D	2×3010 MHz	2048 Kb	2600 (41)	4500 (70)

should be interpreted carefully, and performance on other architectures as well as the optimization potential remain to be studied.

Parallelism. How can LAKE-256 be parallelized? First consider the "medium grain" level: In saltstate, the computation of the L_i's can be parallelized into sixteen branches, since there is no diffusion across word boundaries—the concurrent access to H might however be an obstacle. Due to its large amount of flow dependence, the main function processmessage is not parallelizable, unlike feedforward, which can be split into eight branches. At a finer level, the three internal expressions of f can be computed in parallel (by copying two variables), as well as the two additions of g. This can benefit to the three components.

5 Security

5.1 Introduction

Definitions. For LAKE hash functions, the preimage problem (resp. second preimage) takes as parameter a random digest y (resp. random message and salt of digest y), and the challenge is to find a (distinct) pair message and salt mapping to y. Target second preimage is similar to second preimage except that the two salts must be identical. The collision problem is to find two pairs message and salt with identical digest, and we call a collision *synchronized* if the two salts are identical. The generic (brute force) attack solves preimage with probability ε within $2^{d+\log_2 \varepsilon}$ calls to the hash function. Hellman's time(T)-memory(M) tradeoff is $TM^2 = 2^{2n}$ [16]. Idem for (target) second preimage. Against collision, the generic birthday attack requires $\sqrt{-2\log(1-\varepsilon)} \cdot 2^{d/2}$ calls to the hash function to succeed with probability ε, with negligible memory requirement (thanks to smart variants of Floyd's cycle-finding technique [38, 36]). For evaluating the cryptographic quality of the distributions induced by our hash functions, we suggest to consider the definitions of *pseudo-randomness* and *unpredictability* given by Naor and Reingold [28] for function distributions, which apply as well for salt-indexed families derived from a LAKE instance (in this case the function of the family take as sole input a message).

Conjectures. For all the instances proposed, no method should solve preimage, second preimage, or collision faster than the generic one for any parameters of the problem. We also conjecture pseudo-randomness and unpredictability for families indexed by the s-bit salt. In addition, relaxed problems as pseudo- or near-collision should also be hard (note that our claims concern the *full functions*, not the building blocks individually). On the other hand we expect variants with less than three rounds to be relatively weak—though we have no evidence of it yet.

5.2 One-Wayness and Collision Resistance

One-wayness is achieved mainly thanks to the feedforward operations (in processmessage and feedforward), and to the redundancy in the initial local chaining variable L (in saltstate).

Arguments for collision resistance are given by the structure of processmessage: recall from §3.2 that the local wide-pipe strategy of processmessage makes it injective for a fixed L; we can expect the r-round processmessage to be collision-free as well, or at least have a negligible number of collisions. Synchronized collisions on compress can thus only occur at the ultimate step, feedforward, when the 512-bit state is compressed to a 256-bit chaining variable (for LAKE-256 parameters). Therefore, the objective of the repeated processmessage is to make hard the search for message pairs (M, M') such that

$$\mathsf{feedforward}(\mathsf{processmessage}(L, M, \sigma), H, S, t) = \mathsf{feedforward}(\mathsf{processmessage}(L, M', \sigma), H, S, t),$$

with $L = \mathsf{saltstate}(H, S, t)$. Note that, if synchronized collisions over compress with similar (H, S, t) are hard to find, then the corresponding LAKE instance is target collision resistant. This is because for any collision over the full hash function with similar salts, at least one collision over compress with identical counter exists.

On the other hand, it is easy to find 1-round collisions with (chosen) distinct salts, and identical H: it suffices to take a random message M, and adapt a second one M' to correct changes in the last eight words of the initial L. However, the collision does not persist to subsequent rounds, and seems to have no consequence on the overall security of compress.

5.3 Algebraic Attacks

Traditional algebraic attacks aim at giving an input/output relationship in terms of multivariate equations, then exploiting this system (ideally solving it) to recover secret information. Due to its rather nested structure, LAKE is unlikely to be vulnerable to such attacks: addition carries and chained computation of L ensure an algebraic normal form (ANF) for each Boolean component of maximal degree after one round of processmessage.

Recently, two works aimed at detecting non-uniform randomness in the algebraic structure of several cryptographic primitives [32,31,14]. Their basic idea is to compute all or part of the ANF of some implicit Boolean function, mapping part of the input to part of the output, then applying statistical tests on the distribution of the monomials of the ANF; reference [32] notably claims to distinguish SHA-2 functions from random using so-called Defectoscopy, claiming that *"at least 8 full cycles are required for it to be secure instead of the proposed 4-5"*, *idem* for MD5 and SHA-0/1—unfortunately, the description of the tests and the methodology used in [32] is not precise enough to reproduce the experiments. In [14], comparable tests are used to build distinguishers for the stream ciphers DECIM, Grain, Lex, and Trivium. Against such methods, we applied the following countermeasures:

- *Intensive feedforward*: each round incorporates a complex feedforward operation, such temporal dependency providing highly non-linear relations.

- *Many additions*: compared to SHA-256, we use a large amount of integer additions, and a few XORs (see Table 1), which increases non-linearity through the carries.
- *Large state*: the output of compress is extracted from a large local state combined with a non-linear dependence on the initial H in feedforward (unlike in SHA-2 functions).

We ran a few experiments with the "d-monomial" and "maximum-degree monomial" tests described in [14], and also used in [32]; for input windows of 8 to 20 bits of the salt or of the message, and each of the 256 output bits. Significant deviations where observed for up to two rounds. This bound might be slightly increased by using refined experiments, as apparently employed, but not detailed, in [32].

5.4 Differential Attacks

The essential idea of differential attacks on hash functions [10], as used to break MD5 and SHA-0/1, is to exploit a high-probability input/output differential over some component of the hash function, e.g. under the form of a "perturb-and-correct" strategy for the latter functions, exploiting high-probability linear/non-linear characteristics. A common property of those functions, as well as to the SHA-2 functions, is indeed the behavior of the compression function as a shift register: at each step, a "first" word is updated depending of a message word input, while the other words are shifted. Hence "corrected words" can spread along the state, as explained in the Chabaud-Joux attack on SHA-0 [11]. Moreover, their relatively simple step function allowed to find several high-probability characteristics (see e.g. [10])—the SHA-2 functions made this much more difficult, though have more or less the same structure as their ancestors, and keep a similar number of rounds.

In the design of LAKE-256 and LAKE-512, we applied the following countermeasures against differential attacks:

- *High number of steps*: with respectively 128 and 160 message word inputs in LAKE-256 and LAKE-512, against 64 for MD5 and SHA-256, and 80 for SHA-512 and SHA-0/1. In particular, the function f called 136 and 168 times in compress makes the exploit of linear approximations highly implausible.
- *Nested feedforward*: the r message-dependent internal feedforward operations aim at strengthening the function against differential paths.
- *Internal wide-pipe*: this makes internal collisions unlikely, and the final compression of L to H makes differences in the output much harder to predict.
- *No "shift register"*: in the round function, *all* state words are updated in chain with dependence on a message word, and then undergo a message-independent post-treatment, making any "correction" impossible.
- *Use all operators*: as observed in §3.4, a small set of operations often facilitates differential analysis.

Note that the foregoing features, except the increased number of rounds, do not require extra computation or memory, unlike the use of a recursive message expansion, or of S-boxes.

We can sketch a simple method for finding low-weight 1-round differentials in compress: choose an input difference Δ changing M_{14} and M_{15} such that after the first loop in processmessage, only L_{14} is modified. Consequently, after the second loop changes will occur only in L_{13}, L_{14} and L_{15}, that is, a difference of weight at least 3. However, such low-weight output-difference will persist no further. We discovered no high-probability differential, but a more careful analysis is required.

5.5 Empirical Tests

For completeness, we report some experiments assessing the minimal requirements for a hash function. Note that no statement about preimage or collision resistance should be derived from these results.

Diffusion. To illustrate the difference propagation in LAKE-256, we give visual examples of the diffusion provided by processmessage, after running saltstate with $H = IV$, $S = 0$, $t = 0$. The avalanche effect is suggested by the high number of differences within only two rounds of processmessage. We consider various one-bit differences in random messages, as presented in Fig. 5: the first stripe represents the message difference, and the eight subsequent ones show the differences in the buffer L after each round of processmessage.

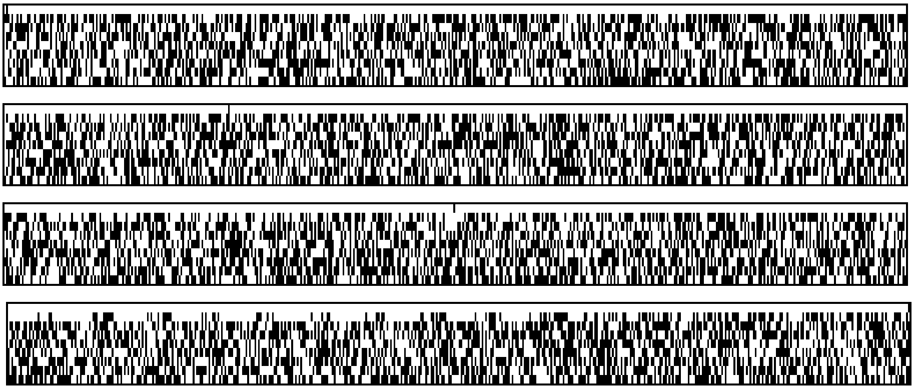


Fig. 5. Diffusion diagrams, for randomly chosen messages and a difference at 2nd, 128th, 256th, and 512th position

The observation that the most-significant bits of the message diffuse less after one round can easily be explained by the algorithm of processmessage. This however has no consequence on the security *per se*, since after only two rounds no kind of regularity seems observable.

Acknowledgments

We would like to thank the reviewers of FSE 2008 who pointed out several issues to improve the paper, and F. Mendel, C. Rechberger, M. Schläffer, and S. Lucks for their analysis of LAKE .

References

1. Andreeva, E., Neven, G., Preneel, B., Shrimpton, T.: Seven-properties-preserving iterated hashing: The RMC construction. Technical Report STVL4-KUL15-RMC-1.0, ECRYPT (2006)
2. Andreeva, E., Neven, G., Preneel, B., Shrimpton, T.: Seven-property-preserving iterated hashing: ROX. In: Kurosawa [21], pp. 130–146
3. Barreto, P., Rijmen, V.: The Whirlpool hashing function. In: First Open NESSIE Workshop (2000)
4. Bellare, M., Goldreich, O., Goldwasser, S.: Incremental cryptography: The case of hashing and signing. In: Desmedt, Y. (ed.) CRYPTO 1994. LNCS, vol. 839, pp. 216–233. Springer, Heidelberg (1994)
5. Bellare, M., Ristenpart, T.: Multi-property-preserving hash domain extension and the EMD transform. In: Lai, Chen [22], pp. 299–314
6. Bernstein, D.J.: The cpucycles library, http://ebats.cr.yp.to/cpucycles.html
7. Bernstein, D.J.: Salsa20. Technical Report 2005/25, ECRYPT eSTREAM, 2005 (2005), http://cr.yp.to/snuffle.html
8. Biham, E., Dunkelman, O.: A framework for iterative hash functions - HAIFA. Cryptology ePrint Archive, Report 2007/278, 2007. In: The second NIST Hash Function Workshop (2006)
9. Biryukov, A. (ed.): FSE 2007. LNCS, vol. 4593. Springer, Heidelberg (2007)
10. De Cannière, C., Rechberger, C.: Finding SHA-1 characteristics: General results and applications. In: Lai, Chen [22], pp. 1–20
11. Chabaud, F., Joux, A.: Differential collisions in SHA-0. In: Krawczyk [20], pp. 56–71
12. Devine, C.: XySSL, http://xyssl.org/code/
13. Dobbertin, H., Bosselaers, A., Preneel, B.: RIPEMD-160: A strengthened version of RIPEMD. In: FSE 1996. LNCS, vol. 1039, pp. 71–82. Springer, Heidelberg (1996)
14. Englund, H., Johansson, T., Turan, M.S.: A framework for chosen IV statistical analysis of stream ciphers. In: Special ECRYPT Workshop – Tools for Cryptanalysis (2007), http://www.impan.gov.pl/BC/Program/conferences/07Crypt-prg.html
15. Halevi, S., Krawczyk, H.: Strengthening digital signatures via randomized hashing. In: Dwork, C. (ed.) CRYPTO 2006. LNCS, vol. 4117, pp. 41–59. Springer, Heidelberg (2006)
16. Hellman, M.: A cryptanalytic time-memory tradeoff. IEEE Transactions on Information Theory 26, 401–406 (1980)
17. Hong, D., Chang, D., Sung, J., Lee, S., Hong, S., Lee, J., Moon, D., Chee, S.: A new dedicated 256-bit hash function: FORK-256. In: Robshaw, M.J.B. (ed.) FSE 2006. LNCS, vol. 4047, pp. 195–209. Springer, Heidelberg (2006)
18. Knudsen, L.R.: SMASH - a cryptographic hash function. In: Gilbert, H., Handschuh, H. (eds.) FSE 2005. LNCS, vol. 3557, pp. 228–242. Springer, Heidelberg (2005)

19. Knudsen, L.R., Rechberger, C., Thomsen, S.S.: The Grindahl hash functions. In: Biryukov [9], pp. 39–57, http://www.ramkilde.com/grindahl/
20. Krawczyk, H. (ed.): CRYPTO 1998. LNCS, vol. 1462. Springer, Heidelberg (1998)
21. Kurosawa, K. (ed.): ASIACRYPT 2007. LNCS, vol. 4833. Springer, Heidelberg (2007)
22. Lai, X., Chen, K. (eds.): ASIACRYPT 2006. LNCS, vol. 4284. Springer, Heidelberg (2006)
23. Lucks, S.: Design principles for iterated hash functions. Cryptology ePrint Archive, Report 2004/253 (2004)
24. Lucks, S.: A failure-friendly design principle for hash functions. In: Roy, B.K. (ed.) ASIACRYPT 2005. LNCS, vol. 3788, pp. 474–494. Springer, Heidelberg (2005)
25. Matusiewicz, K., Peyrin, T., Billet, O., Contini, S., Pieprzyk, J.: Cryptanalysis of FORK-256. In: Biryukov [9], pp. 19–38, http://www.ics.mq.edu.au/~kmatus/FORK/
26. Mendel, F., Rechberger, C., Schläffer, M.: Collisions for round-reduced LAKE (submitted, 2008)
27. Mendel, F., Rijmen, V.: Cryptanalysis of the Tiger hash function. In: Kurosawa [21], pp. 536–550.
28. Naor, M., Reingold, O.: From unpredictability to indistinguishability: A simple construction of pseudo-random functions from MACs (extended abstract). In: Krawczyk [20], pp. 267–282
29. NIST. FIPS 180-2 secure hash standard (2002)
30. NIST. Cryptographic hash project (2007), http://www.nist.gov/hash-competition
31. O'Neil, S.: Algebraic structure defectoscopy, http://defectoscopy.com/
32. O'Neil, S.: Algebraic structure defectoscopy. In: Special ECRYPT Workshop – Tools for Cryptanalysis (2007), http://www.impan.gov.pl/BC/Program/conferences/07Crypt-prg.html
33. Pal, P., Sarkar, P.: PARSHA-256 - a new parallelizable hash function and a multithreaded implementation. In: Johansson, T. (ed.) FSE 2003. LNCS, vol. 2887, pp. 347–361. Springer, Heidelberg (2003)
34. Peyrin, T.: Cryptanalysis of Grindahl. In: Kurosawa [21], pp. 551–567.
35. Pramstaller, N., Rechberger, C., Rijmen, V.: Breaking a new hash function design strategy called SMASH. In: Preneel, B., Tavares, S.E. (eds.) SAC 2005. LNCS, vol. 3897, pp. 233–244. Springer, Heidelberg (2006)
36. Quisquater, J.-J., Delescaille, J.-P.: How easy is collision search? Application to DES (extended summary). In: Quisquater, J.-J., Vandewalle, J. (eds.) EUROCRYPT 1989. LNCS, vol. 434, pp. 429–434. Springer, Heidelberg (1990)
37. Schneier, B., Kelsey, J., Whiting, D., Wagner, D., Hall, C., Ferguson, N.: The Twofish Encryption Algorithm. Wiley, Chichester (1999)
38. Sedgewick, R., Szymanski, T.G., Yao, A.C.-C.: The complexity of finding cycles in periodic functions. SIAM Journal of Computing 11(2), 376–390 (1982)
39. Wheeler, D.J., Needham, R.M.: TEA, a tiny encryption algorithm. In: Preneel, B. (ed.) FSE 1994. LNCS, vol. 1008, pp. 363–366. Springer, Heidelberg (1995)
40. Whiting, D., Schneier, B., Lucks, S., Muller, F.: Phelix - fast encryption and authentication in a single cryptographic primitive. Technical Report 2005/20, ECRYPT eSTREAM (2005)

A Constants

For LAKE-256, IV corresponds to the first 64 hexadecimal digits of π, and the constants to the 65-th to the 192-th digits[3]:

$IV_0 =$ 243F6A88	$IV_2 =$ 13198A2E	$IV_4 =$ A4093822	$IV_6 =$ 082EFA98
$IV_1 =$ 85A308D3	$IV_3 =$ 03707344	$IV_5 =$ 299F31D0	$IV_7 =$ EC4E6C89

$C_0 =$ 452821E6	$C_4 =$ C0AC29B7	$C_8 =$ 9216D5D9	$C_{12} =$ 2FFD72DB
$C_1 =$ 38D01377	$C_5 =$ C97C50DD	$C_9 =$ 8979FB1B	$C_{13} =$ D01ADFB7
$C_2 =$ BE5466CF	$C_6 =$ 3F84D5B5	$C_{10} =$ D1310BA6	$C_{14} =$ B8E1AFED
$C_3 =$ 34E90C6C	$C_7 =$ B5470917	$C_{11} =$ 98DFB5AC	$C_{15} =$ 6A267E96

For LAKE-512, IV corresponds to the first 128 hexadecimal digits of π ending trillion-th, and the constants to the 129-th to the 384-th digits.

$IV_0 =$ 57F5C7D088813AFC	$IV_4 =$ F92F3FFEB7790C39
$IV_1 =$ 13908A7C25E945C0	$IV_5 =$ 428D3FD1A930A4EE
$IV_2 =$ B273D634AF4635AB	$IV_6 =$ A66C46E2B3255458
$IV_3 =$ B8E6A0E2AE025B8F	$IV_7 =$ F2AC54FEDE1EC2EA

$C_0 =$ 0769441AD54C789F	$C_8 =$ 4623A40AB23A2E02
$C_1 =$ 3CB62BB721C2746E	$C_9 =$ A43BA7CDFC9BCF82
$C_2 =$ 1BE973B3FF6C5EDE	$C_{10} =$ D6AEBF43FB266C5E
$C_3 =$ D9883F666CD37F6B	$C_{11} =$ 139363097AAB1247
$C_4 =$ 2A9572193E06AA68	$C_{12} =$ 2A53B4E0A95CAA01
$C_5 =$ 8AB87CA9222605F2	$C_{13} =$ 8D1770714B749520
$C_6 =$ 3B43E1D7013CEAC5	$C_{14} =$ B3BC88DB689CA207
$C_7 =$ DF6534E1E77E037E	$C_{15} =$ C46EF39031B3E5A5

B Test Values

For LAKE-256, compress(Null input) =

```
C5EB97EC 704D4816 5A1714E3 549343B4 18831B53 2FB84D85 E304A0A4 73CB9E03.
```

For LAKE-512, compress(Null input)=

```
804829AB81DA589B E9205F12A4EE3666 D23D5574793C9C32 4DB7387F53795476
653D40810DC4A3AA F14D3A5E8D14F043 9904191ADE724751 C9D033C934C9229E.
```

[3] Hexadecimal π digits are copied from `http://www.super-computing.org/pi-hexa_current.html`

SWIFFT: A Modest Proposal for FFT Hashing[⋆]

Vadim Lyubashevsky[1], Daniele Micciancio[1], Chris Peikert[2,⋆⋆],
and Alon Rosen[3]

[1] University of California at San Diego
[2] SRI International
[3] IDC Herzliya

Abstract. We propose SWIFFT, a collection of compression functions that are highly parallelizable and admit very efficient implementations on modern microprocessors. The main technique underlying our functions is a novel use of the *Fast Fourier Transform* (FFT) to achieve "diffusion," together with a linear combination to achieve compression and "confusion." We provide a detailed security analysis of concrete instantiations, and give a high-performance software implementation that exploits the inherent parallelism of the FFT algorithm. The throughput of our implementation is competitive with that of SHA-256, with additional parallelism yet to be exploited.

Our functions are set apart from prior proposals (having comparable efficiency) by a supporting asymptotic *security proof*: it can be formally proved that finding a collision in a randomly-chosen function from the family (with noticeable probability) is at least as hard as finding short vectors in cyclic/ideal lattices in the *worst case*.

1 Introduction

In cryptography, there has traditionally been a tension between efficiency and rigorous security guarantees. The vast majority of proposed cryptographic hash functions have been designed to be highly efficient, but their resilience to attacks is based only on intuitive arguments and validated by intensive cryptanalytic efforts. Recently, new cryptanalytic techniques [29,30,4] have started casting serious doubts both on the security of these specific functions and on the effectiveness of the underlying design paradigm.

On the other side of the spectrum, there are hash functions having rigorous asymptotic proofs of security (i.e., security reductions), assuming that various computational problems (such as the discrete logarithm problem or factoring large integers) are hard to solve on the average. Unfortunately, all such proposed hash functions have had computation cost comparable to typical public key cryptographic operations, making them unattractive from a practical point of view.

[⋆] **Mod·est**, adj.: Marked by simplicity.

[⋆⋆] Supported by the National Science Foundation under Grants CNS-0716786 and CNS-0749931. Any opinions, findings, and conclusions or recommedations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

K. Nyberg (Ed.): FSE 2008, LNCS 5086, pp. 54–72, 2008.

1.1 Our Proposal: SWIFFT

We propose the *SWIFFT* collection of compression functions, and give a high-performance software implementation. SWIFFT is very appealing and intuitive from a traditional design perspective, and, at the same time, achieves the robustness and reliability benefits of *provable* asymptotic security under a mild computational assumption. The functions correspond to a simple algebraic expression over a certain polynomial ring, as described in detail in Section 2.1. Here we describe a high-level algorithm for the fast evaluation of a SWIFFT compression function.

The algorithm takes as input a binary string of length mn (for suitable parameters m, n), which is viewed as an $n \times m$ binary matrix $(x_{i,j}) \in \{0,1\}^{n \times m}$. It then performs the following two steps, where all operations are performed in $\mathbb{Z}_p$ for an appropriate modulus p:

1. The input matrix $(x_{i,j})$ is first processed by multiplying the ith row by ω^{i-1} for $i = 1, \ldots, n$ (where $\omega \in \mathbb{Z}_p$ is an appropriate fixed element). Then the *Fast Fourier Transform* (FFT) is computed (over $\mathbb{Z}_p$) on each column $j = 1, \ldots, m$:

$$(y_{1,j} \ , \ \ldots \ , \ y_{n,j}) = \mathrm{FFT}(\omega^0 \cdot x_{1,j} \ , \ \ldots \ , \ \omega^{n-1} \cdot x_{n,j}).$$

   We remark that this operation is easy to invert, and is performed to achieve "diffusion," i.e., to mix the input bits of every column.
2. A linear combination is then computed across each row $i = 1, \ldots, n$:

$$z_i = a_{i,1} \cdot y_{i,1} + \cdots + a_{i,m} \cdot y_{i,m} = \sum\nolimits_{j=1}^{m} a_{i,j} \cdot y_{i,j},$$

   where the coefficients $a_{i,j} \in \mathbb{Z}_p$ are fixed as part of the function description. This operation compresses the input, achieving "confusion."

The output is the vector $(z_1, \ldots, z_n) \in \mathbb{Z}_p^n$.

Consider an attempt to invert the function, i.e., to find some input $(x_{i,j})$ that evaluates to a given output $(z_1, \ldots, z_n)$. Viewed independently, each linear equation $z_i = \sum_{j=1}^{m} a_{i,j} \cdot y_{i,j}$ on the rows admits a large number of easily-computed solutions. However, there are strong dependencies among the equations. In particular, every column $(y_{1,j}, \ldots, y_{n,j})$ is constrained to be the result of applying Step 1 to an n-dimensional *binary* vector $(x_{1,j}, \ldots, x_{n,j}) \in \{0,1\}^n$.

Perhaps surprisingly, these constraints turn out to be sufficient to guarantee asymptotically that the SWIFFT functions are *provably* one-way and collision-resistant. More precisely, the family admits a very strong security reduction: finding collisions on the average (when the coefficients $a_{i,j}$ are chosen at random in $\mathbb{Z}_p$) with any noticeable probability is at least as hard as solving an underlying mathematical problem on certain kinds of *point lattices* in the *worst case*. This claim follows from the fact that the SWIFFT functions are a special case of the *cyclic/ideal lattice*-based functions of [18,21,16].

SWIFFT's simple design has a number of other advantages. First, it also enables unconditional proofs of a variety of *statistical* properties that are desirable in many applications of hash functions, both in cryptography and in other domains. Second, its underlying mathematical structure is closely related to well-studied cryptographic problems, which permits easy understanding and analysis of concrete instantiations. Third, it is extremely parallelizable, and admits software implementations with throughput comparable to (or even exceeding) the SHA-2 family on modern microprocessors.

While SWIFFT satisfies many desirable cryptographic and statistical properties, we caution that it was not designed to be an "all-purpose" cryptographic hash function. For example, it is not (by itself) a pseudorandom function, and would not be a suitable instantiation of a random oracle. (See Section 4 for more details.) In addition, while the concrete parameters were chosen so as to resist all known feasible attacks, SWIFFT does not achieve full "birthday bound" security of $2^{n/2}$ for collision attacks with an n-bit output, nor 2^n security for inversion attacks. (See Section 5 for more details.)

1.2 Related Work

Using the Fast Fourier Transform (FFT) as a building block in hash functions is not new. For example, Schnorr *et al* proposed a variety of FFT-based hash functions [24,25,26], which unfortunately were subsequently cryptanalyzed and shown to be insecure [9,2,27]. Our compression functions are set apart from previous work by the way that the FFT is used, and the resulting proof of security. Namely, while in previous work [24,25,26] the FFT was applied to *unrestricted* input vectors $(x_1, \ldots, x_n) \in \mathbb{Z}_p^n$, here we require the input values x_i to be bits. This introduces non-linear constraints on the output values of the FFT operation, a fact that plays a fundamental role both in our theoretical proof of security, as well as on the analysis of our concrete functions. Our novel use of FFT may be of independent interest, and might find other applications in cryptographic design.

The subset-sum and knapsack problems have long ago been suggested as foundations for compression functions, e.g., by Damgård [10]. Unfortunately, these functions are only efficient in small dimensions, at which point lattice-based attacks [14] and other forms of cryptanalysis [7] become possible.

An important ingredient in the conceptual design of our functions (and associated proof of security) is the use of lattices with special structure as an underlying mathematical problem. Special classes of lattices (with closely related, but somewhat different structure than ours) also have been used before in practical constructions (most notably, the NTRU encryption scheme [13] and LASH hash function [3]), but without any security proofs.

Most closely related to our work is the theoretical study initiated by Ajtai [1] of subset sum-like cryptographic functions that are provably secure under worst-case assumptions for lattice problems. Ajtai's work and subsequent improvements [11,6,17,19] do not lead to very efficient implementations, mostly because of the huge size of the function description and slow evaluation time (which grow quadratically in the security parameter). A first step toward bridging the gap between theoretical constructions and practical functions was taken by Micciancio [18], who proposed using lattices with special structure (namely, cyclic lattices) and showed how they lead to cryptographic functions that have provable worst-case hardness and also admit fast implementations using FFT. The main limitation of the functions proposed in [18] was the notion of security achieved: they are provably one-way (under a worst-case assumption on cyclic lattices), but not collision resistant. Peikert and Rosen [21] and Lyubashevsky and Micciancio [16] then modified and generalized the function originally proposed in [18] to achieve collision resistance.

From a theoretical point of view, the SWIFFT functions proposed in this paper are equivalent to and inherit all provable security features from the cyclic/ideal hash functions of [21,16]. But differently from [18,21,16], the emphasis in this paper is on practical implementation issues, and the construction of concrete instances and variants of those functions that enjoy very efficient implementation from a practical point of view. For a deeper understanding of the theoretical ideas underlying the proofs of security of our compression functions, we refer the reader to [18,21,16].

2 SWIFFT Compression Functions

In this section, we describe an algebraic expression that is the underlying foundation of the SWIFFT functions, and how it is related to the FFT-based algorithm described in Section 1.1. We then propose a set of concrete parameters on which our implementation and security analysis are based.

2.1 Algebraic Description

The SWIFFT functions correspond to a simple algebraic expression over a certain polynomial ring. A *family* of SWIFFT functions is described by three main parameters: let n be a power of 2, let $m > 0$ be a small integer, and let $p > 0$ be a modulus (not *necessarily* prime, though we will soon see that certain prime p will be convenient). Define R to be the ring $R = \mathbb{Z}_p[\alpha]/(\alpha^n + 1)$, i.e., the ring of polynomials (in α) having integer coefficients, modulo p and $\alpha^n + 1$. Any element of R may therefore be written as a polynomial of degree $< n$ having coefficients in $\mathbb{Z}_p = \{0, \ldots, p-1\}$.

A particular *function* in the family is specified by m fixed elements $\mathbf{a}_1, z \ldots, \mathbf{a}_m \in R$ of the ring R, called "multipliers." The function corresponds to the following expression over the ring R:

$$\sum\nolimits_{i=1}^{m} (\mathbf{a}_i \cdot \mathbf{x}_i) \quad \in \quad R, \tag{1}$$

where $\mathbf{x}_1, \ldots, \mathbf{x}_m \in R$ are polynomials having *binary* coefficients, and corresponding to the binary input of length mn.

To compute the above expression, the main bottleneck is in computing the polynomial products $\mathbf{a}_i \cdot \mathbf{x}_i$ over R. It is well-known that the *Fast Fourier Transform* (FFT) provides an $O(n \log n)$-time algorithm that can be used for multiplying polynomials of degree $< n$. The multiplication algorithm starts by using the FFT to compute (all at once) the *Fourier coefficients* of each polynomial, i.e., the values on all the $2n$th roots of unity over the complex field $\mathbb{C}$. It then multiplies the respective Fourier coefficients of the two polynomials, and finally interpolates back to a degree $< 2n$ polynomial via an inverse FFT.

Because we are working modulo p and $\alpha^n + 1$, there is an even more convenient and efficient method for computing the polynomial products in the ring R. Suppose that p is prime and $p - 1$ is a multiple of $2n$. Then $\mathbb{Z}_p$ is a field, and it contains a multiplicative subgroup of order $2n$ whose elements are all the $2n$th roots of unity in $\mathbb{Z}_p$ (i.e., the roots of the polynomial $\alpha^{2n} - 1 \bmod p$). Let $\omega \in \mathbb{Z}_p$ be some generator of this subgroup, i.e., an element of order $2n$. The n *odd* powers $\omega^1, \omega^3, \ldots, \omega^{2n-1}$ are exactly the *primitive* $2n$th roots of unity, i.e., the roots of $\alpha^n + 1$.

In order to compute a polynomial product $\mathbf{a}_i \cdot \mathbf{x}_i$ modulo p and $\alpha^n + 1$, it suffices to compute only the n *primitive* Fourier coefficients of $\mathbf{a}_i$ and $\mathbf{x}_i$, i.e., the values $\mathbf{a}_i(\omega^1), \mathbf{a}_i(\omega^3), \ldots, \mathbf{a}_i(\omega^{2n-1})$, and likewise for $\mathbf{x}_i$. The primitive coefficients can be computed all at once by preprocessing the input and then applying an n-dimensional FFT (which uses half the space), as described in the algorithm from Section 1.1. Furthermore, because the field $\mathbb{Z}_p$ has roots of unity, the FFT can be performed over $\mathbb{Z}_p$ using the nth primitive root of unity ω^2, instead of over $\mathbb{C}$.[1]

In addition to using an FFT, other significant optimizations are possible when computing Expression (1). First, because the multipliers $\mathbf{a}_i$ are fixed in advance and determined uniquely by their primitive Fourier coefficients, we can simply store and work with their Fourier representation. Additionally, because the FFT is linear and a bijection, there is no need to even apply an *inverse* FFT. In other words, the value of Expression (1) is correctly and uniquely determined by summing the Fourier representations of $\mathbf{a}_i \cdot \mathbf{x}_i$. Combining all these observations, we are left with the high-level algorithm as described in Section 1.1, which we implement (using additional optimizations) in Section 3.

2.2 Concrete Parameters

In this paper we primarily study one family of SWIFFT compression functions, obtained by choosing concrete values for the parameters n, m, and p as follows:

$$n = 64, \quad m = 16, \quad p = 257.$$

For these parameters, any fixed compression function in the family takes a binary input of length $mn = 1024$ bits (128 bytes), to an output in the range $\mathbb{Z}_p^n$, which

[1] Performing an FFT over $\mathbb{Z}_p$ rather than $\mathbb{C}$ is often called a *number theoretic transform* (NTT) in the literature; however, we will retain the FFT terminology due to broad familiarity.

has size $p^n = 257^{64} \approx 2^{512}$. An output in $\mathbb{Z}_p^n$ can easily be represented using 528 bits (66 bytes). Other unambiguous representations (using > 512 bits) are also possible; the representation does not affect security.

We now briefly explain our choice of parameters. The first consideration is the security of the compression function. As we will explain in the security analysis of Section 5, the function corresponds to a subset-sum from mn bits to roughly $n \lg p$ bits. We first set the constraints $mn = 1024$ and $n \lg p \approx 512$, because solving such subset-sum problems appears to be intractable. In order for our proofs of security to go through, we also need the polynomial $\alpha^n + 1$ to be irreducible over $\mathbb{Z}[\alpha]$, which is true if and only if n is a power of 2. (If a *reducible* polynomial is used, actual attacks can become possible, as we show in Section 5.3 for similar functions in the literature.)

Next, we optimize the running time and space of the function by choosing n to be relatively large, and p and m to be small, subject to the above constraints. As discussed above, the Fast Fourier Transform is most efficiently and conveniently computed when p is prime and $p - 1$ is a multiple of $2n$.

Finally, to fix one concrete function from the family, the multipliers $\mathbf{a}_i$ should be chosen uniformly and independently at random from the ring R; this is equivalent to choosing their primitive Fourier coefficients uniformly and independently at random from $\mathbb{Z}_p$. We note that the multipliers (or their Fourier coefficients) should be chosen using "trusted randomness," otherwise it may be possible to embed a "backdoor" in the resulting function. For example, one might derive the multipliers using some deterministic transformation on the digits of π.

3 Implementation

Our implementation uses two main techniques for achieving high performance, both relating to the structure of the Fast Fourier Transform (FFT) algorithm. The first observation is that the input to the FFT is a *binary* vector, which limits the number of possible input values (when restricting our view to a small portion of the input). This allows us to precompute and store the results of several initial iterations of the FFT in a lookup table. The second observation is that the FFT algorithm consists of operations repeated in parallel over many pieces of data, for which modern microprocessors have special-purpose instruction sets.

Recall the parameters $n = 64$, $m = 16$, and modulus $p = 257$. Let ω be a 128th root of unity in $\mathbb{Z}_p = \mathbb{Z}_{257}$, i.e., an element of order $128 = 2n$. (We will see later that it is convenient to choose $\omega = 42$, but most of the discussion is independent from the choice of ω.)

The compression function takes an $mn = 1024$-bit input, viewed as $m = 16$ binary vectors $\mathbf{x}_0, \ldots, \mathbf{x}_{15} \in \{0,1\}^{64}$. (For convenience, entries of a vector or sequence are numbered starting from 0 throughout this section.) The function first processes each vector $\mathbf{x}_j$, multiplying its ith entry by ω^i (for $i = 0, \ldots, 63$), and then computing the Fourier transform of the resulting vector using ω^2 as a 64th root of unity. More precisely, each input vector $\mathbf{x}_j \in \{0,1\}^{64}$ is mapped to

$\mathbf{y}_j = F(\mathbf{x}_j)$, where $F\colon \{0,1\}^{64} \to \mathbb{Z}_{257}^{64}$ is the function

$$F(\mathbf{x})_i = \sum_{k=0}^{63} (x_k \cdot \omega^k) \cdot (\omega^2)^{i \cdot k} = \sum_{k=0}^{63} x_k \cdot \omega^{(2i+1)k}. \tag{2}$$

The final output $\mathbf{z}$ of the compression function is then obtained by computing 64 distinct linear combinations (modulo 257) across the ith entries of the 16 $\mathbf{y}_j$ vectors:

$$z_i = \sum_{j=0}^{15} a_{i,j} \cdot y_{i,j} \pmod{257},$$

where the $a_{i,j} \in \mathbb{Z}_{257}$ are the primitive Fourier coefficients of the fixed multipliers.

Computing F. The most expensive part of the computation is clearly the computation of the transformation F on the 16 input vectors $\mathbf{x}_j$, so we first focus on the efficient computation of F. Let $\mathbf{y} = F(\mathbf{x}) \in \mathbb{Z}_{257}^{64}$ for some $\mathbf{x} \in \{0,1\}^{64}$. Expressing the indices i, k from Equation (2) in octal as $i = i_0 + 8i_1$ and $k = k_0 + 8k_1$ (where $j_0, j_1, k_0, k_1 \in \{0, \ldots, 7\}$), and using $\omega^{128} = 1 \pmod{257}$, the ith component of $\mathbf{y} = F(\mathbf{x})$ is seen to equal

$$\begin{aligned} y_{i_0+8i_1} &= \sum_{k_0=0}^{7} (\omega^{16})^{i_1 \cdot k_0} \left(\omega^{(2i_0+1)k_0} \cdot \sum_{k_1=0}^{7} \omega^{8k_1(2i_0+1)} \cdot x_{k_0+8k_1} \right) \\ &= \sum_{k_0=0}^{7} (\omega^{16})^{i_1 \cdot k_0} \left(m_{k_0,i_0} \cdot t_{k_0,i_0} \right), \end{aligned}$$

where $m_{k_0,i_0} = \omega^{(2i_0+1)k_0}$ and $t_{k_0,i_0} = \sum_{k_1=0}^{7} \omega^{8k_1(2i_0+1)} x_{k_0+8k_1}$. Our first observation is that each 8-dimensional vector $\mathbf{t}_{k_0} = (t_{k_0,0}, t_{k_0,1}, \ldots, t_{k_0,7})$ can take only 256 possible values, depending on the corresponding input bits $x_{k_0}, x_{k_0+8}, \ldots, x_{k_0+8 \cdot 7}$. Our implementation parses each 64-bit block of the input as a sequence of 8 bytes $X_0, \ldots, X_7$, where $X_{k_0} = (x_{k_0}, x_{k_0+8}, \ldots, x_{k_0+8 \cdot 7}) \in \{0,1\}^8$, so that each vector $\mathbf{t}_{k_0}$ can be found with just a single table look-up operation $\mathbf{t}_{k_0} = T(X_{k_0})$, using a table T with 256 entries. The multipliers $\mathbf{m}_{k_0} = (m_{k_0,0}, \ldots, m_{k_0,7})$ can also be precomputed.

The value $\mathbf{y} = F(\mathbf{x})$ can be broken down as 8 (8-dimensional) vectors

$$\mathbf{y}_{i_1} = (y_{8i_1}, y_{8i_1+1}, \ldots, y_{8i_1+7}) \in \mathbb{Z}_{257}^{8}.$$

Our second observation is that, for any $i_0 = 0, \ldots, 7$, the i_0th component of $\mathbf{y}_{i_1}$ depends only on the i_0th components of $\mathbf{m}_{k_0}$ and $\mathbf{t}_{k_0}$. Moreover, the operations performed for every coordinate are exactly the same. This permits parallelizing the computation of the output vectors $\mathbf{y}_0, \ldots, \mathbf{y}_7$ using SIMD (single-instruction multiple-data) instructions commonly found on modern microprocessors. For example, Intel's microprocessors (starting from the Pentium 4) include a set of so-called SSE2 instructions that allow operations on a set of special registers each holding an 8-dimensional vector with 16-bit (signed) integer components.

We only use the most common SIMD instructions (e.g., component-wise addition and multiplication of vectors), which are also found on most other modern microprocessors, e.g., as part of the AltiVec SIMD instruction set of the Motorola G4 and IBM G5 and POWER6. In the rest of this section, operations on 8-dimensional vectors like $\mathbf{m}_{k_0}$ and $\mathbf{t}_{k_0}$ are interpreted as scalar operations applied component-wise to the vectors, possibly in parallel using a single SIMD instruction.

Going back to the computation of $F(\mathbf{x})$, the output vectors $\mathbf{y}_{i_1}$ can be expressed as

$$\mathbf{y}_{i_1} = \sum_{k_0=0}^{7} (\omega^{16})^{i_1 \cdot k_0} (\mathbf{m}_{k_0} \cdot \mathbf{t}_{k_0}).$$

Our third observation is that the latter computation is just a sequence of 8 component-wise multiplications $\mathbf{m}_{k_0} \cdot \mathbf{t}_{k_0}$, followed by a single 8-dimensional Fourier transform using ω^{16} as an 8th root of unity in $\mathbb{Z}_{257}$. The latter can be efficiently implemented using a standard FFT network consisting of just 12 additions, 12 subtractions and 5 multiplications.

Optimizations relating to $\mathbb{Z}_{257}$. One last source of optimization comes from two more observations that are specific to the use of 257 as a modulus, and the choice of $\omega = 42$ as a 128th root of unity. One observation is that the root used in the 8-dimensional FFT computation equals $\omega^{16} = 2^2 \pmod{257}$. So, multiplication by (ω^{16}), $(\omega^{16})^2$ and $(\omega^{16})^3$, as required by the FFT, can be simply implemented as left bit-shift operations (by 2, 4, and 6 positions, respectively). Moreover, analysis of the FFT network shows that modular reduction can be avoided (without the risk of overflow using 16-bit arithmetic) for most of the intermediate values. Specifically, in our implementation, modular reduction is performed for only 3 of the intermediate values. The last observation is that, even when necessary to avoid overflow, reduction modulo 257 can be implemented rather cheaply and using common SIMD instructions, e.g., a 16-bit (signed) integer can be reduced to the range $\{-127, \ldots, 383\}$ using $x \equiv (x \wedge 255) - (x \gg 8) \bmod 257$, where $\wedge$ is the bit-wise "and" operation, and $\gg 8$ is a right-shift by 8 bits.

Summary and performance. In summary, function F can be computed with just a handful of table look-ups and simple SIMD instructions on 8 dimensional vectors. The implementation of the remaining part of the computation of the compression function (i.e., the scalar products between $y_{i,j}$ and $a_{i,j}$) is straightforward, keeping in mind that this part of the computation can also be parallelized using SIMD instructions, and that reduction modulo 257 is rarely necessary during the intermediate steps of the computation due to the use of 16-bit (or larger) registers.

We implemented and tested our function on a 3.2GHz Intel Pentium 4. The implementation was written in C (using the Intel intrinsics to instruct the compiler to use SSE2 instructions), and compiled using gcc version 4.1.2 (compiler flags -O3) on a PC running under Linux kernel 2.6.18. Our tests show that our basic compression function can be evaluated in 1.5 μs on the above system, yielding a throughput close to 40 MB/s in a standard chaining mode of operation. For comparison, we tested SHA256 on the same system using the highly optimized implementation in `openssl` version 0.9.8 (using the `openssl speed` benchmark), yielding a throughput of 47 MB/s when run on 8KB blocks.

Further optimizations. We remark that our implementation does not yet take advantage of all the potential for parallelism. In particular, we only exploited SIMD-level parallelism in individual evaluations of the transformation function F. Each evaluation of the compression function involves 16 applications of F, and subsequent multiplication of the result by the coefficients $a_{i,j}$. These 16 computations are completely independent, and can be easily executed in parallel on a multicore microprocessor. Our profiling data shows that the FFT computations and multiplication by $a_{i,j}$ currently account for about 90% of the running time. So, as multicore processors become more common, and the number of cores available on a processor increases, one can expect the speed of our function to grow almost proportionally to the number of cores, at least up to 16 cores. Finally, we point out that FFT networks are essentially "optimally parallelizable," and that our compression function has extremely small circuit depth, allowing it to be computed extremely fast in customized hardware.

4 Properties of SWIFFT

Here we review a number of statistical and cryptographic properties that are often desirable in hash functions, and discuss which properties our functions do and do not satisfy.

4.1 Statistical Properties

Here we review a number of many well-known and useful *statistical* properties that are often desirable in a family of hash functions, in both cryptographic and non-cryptographic applications (e.g., hash tables, randomness generation). All of these statistical properties can be proved *unconditionally*, i.e., they do not rely on any unproven assumptions about any computational problems.

Universal hashing. A family of functions is called *universal* if, for any fixed distinct x, x', the probability (over the random choice of f from the family) that $f(x) = f(x')$ is the inverse of the size of the range. It is relatively straightforward to show that our family of compression functions is universal (this property is used implicitly in the proofs for the statistical properties below).

Regularity. A function f is said to be *regular* if, for an input x chosen uniformly at random from the domain, the output $f(x)$ is distributed uniformly over the range. More generally, the function is ϵ-regular if its output distribution is within statistical distance (also known as variation distance) ϵ from uniform over the range. The only randomness is in the choice of the input.

As first proved in [18], our family of compression functions is regular in the following sense: all but an ϵ fraction of functions f from the family are ϵ-regular, for some negligibly small ϵ. The precise concrete value of ϵ is determined by the particular parameters (n, m, p) of the family.

Randomness extraction. Inputs to a hash function are often not chosen *uniformly* from the domain, but instead come from some non-uniform "real-world" distribution. This distribution is usually unknown, but may reasonably be assumed to have some amount of uncertainty, or *min-entropy*. For hash tables and related applications, it is usually desirable for the outputs of the hash function to be distributed uniformly (or as close to uniformly as possible), even when the inputs are not uniform. Hash functions that give such guarantees are known as *randomness extractors*, because they "distill" the non-uniform randomness of the input down to an (almost) uniformly-distributed output. Formally, randomness extraction is actually a property of a *family* of functions, from which one function is chosen at random (and obliviously to the input).

The proof of regularity for our functions can be generalized to show that they are also good randomness extractors, for input distributions having enough min-entropy.

4.2 Cryptographic Properties

Here we discuss some well-known properties that are often desirable in cryptographic applications of hash functions, e.g., digital signatures. Under relatively mild assumptions, our functions satisfy several (but not all) of these cryptographic properties. (For precise definitions, see, e.g., [23].)

Informally, a function f is said to *one-way* if, given the value $y = f(x)$ for an x chosen uniformly at random from the domain, it is infeasible for an adversary to find *any* x' in the domain such that $f(x') = y$. It is *second preimage resistant* if, given both x and $y = f(x)$ (where x is again random), it is infeasible to find a *different* $x' \neq x$ such that $f(x') = y$. These notions also apply to families of functions, where f is chosen at random from the family.

A family of functions is *target collision resistant* (also called *universal one-way*) if it is infeasible to find a second preimage of x under f, where x is first chosen by the adversary (instead of at random) and then the function f is chosen at random from the family. Finally, the family is fully *collision resistant* if it is infeasible for an adversary, given a function f chosen at random from the family, to find distinct x, x' such that $f(x) = f(x')$.

For functions that compress their inputs, the notions above are presented in increasing order of cryptographic strength. That is, collision resistance implies target collision resistance, which in turn implies second preimage resistance, which in turn implies one-wayness. All of the above notions are *computational*, in that they refer to the infeasibility (i.e., computational difficulty) of solving some cryptographic problem. However, the concrete effort required to violate these security properties (i.e., the meaning of "infeasible") will vary depending on the specific security notion under consideration, and is discussed in more detail below in Section 5.

As shown in [21,16], our family of compression functions is *provably* collision resistant (in an asymptotic sense), under a relatively mild assumption about the *worst-case* difficulty of finding short vectors in cyclic/ideal lattices. This in turn implies that the family is also one-way and second preimage resistant. In Section 5.1, we give a detailed discussion and interpretation of the security proofs. In Section 5.2, we discuss the best known attacks on the cryptographic properties of our concrete functions, and give estimates of their complexity.

4.3 Properties *Not Satisfied* by SWIFFT

For general-purpose cryptographic hash functions and in certain other applications, additional properties are often desirable. We discuss some of these properties below, but stress that our functions *do not satisfy these properties*, nor were they intended or designed to.

Pseudorandomness. Informally, a family of functions is *pseudorandom* if a randomly-chosen function from the family "acts like" a *truly random* function in its input-output behavior. More precisely, given (adaptive) *oracle access* to a function f, no adversary can efficiently distinguish between the case where (1) f is chosen at random from the given family, and (2) every output of f is uniformly random and independent of all other outputs. (The formal definition is due to [12].) We stress that the adversary's view of the function is limited to oracle access, and that the particular choice of the function from the family is kept secret.

Our family of functions is *not* pseudorandom (at least as currently defined), due to linearity. Specifically, for any function f from our family and any two inputs x_1, x_2 such that $x_1 + x_2$ is also a valid input, we have $f(x_1) + f(x_2) = f(x_1 + x_2)$. This relation is very unlikely to hold for a random function, so an adversary can easily distinguish our functions from random functions by querying the inputs x_1, x_2, and $x_1 + x_2$. However, this homomorphism might actually be considered as a useful *feature* of the function in certain applications (much like homomorphic encryption).

With additional techniques, it may be possible to construct a family of pseudorandom functions (under suitable lattice assumptions) using similar design ideas.

Random oracle behavior. Intuitively, a function is said to behave like a *random oracle* if it "acts like" a truly random function. This notion differs from pseudorandomness in that the function is *fixed* and *public*, i.e., its entire description is known to the adversary. Though commonly used, the notion of "behaving like a random oracle" cannot be defined precisely in any meaningful or achievable way. Needless to say, we do not claim that our functions behave like a random oracle.

5 Security Analysis

In this section, we interpret our asymptotic proofs of security for collision-resistance and the other claimed cryptographic properties. We then consider cryptanalysis of the functions for our specific choice of parameters, and review the best known attacks to determine concrete levels of security.

5.1 Interpretation of Our Security Proofs

As mentioned above, an asymptotic proof of one-wayness for SWIFFT was given in [18], and an asymptotic proof of collision-resistance (a stronger property) was given independently in [21] and [16]. As in most cryptography, security proofs must rely on some precisely-stated (but as-yet unproven) assumption. Our assumption, stated informally, is that finding relatively short nonzero vectors in n-dimensional *ideal lattices* over the ring $\mathbb{Z}[\alpha]/(\alpha^n + 1)$ is infeasible *in the worst case*, as n increases. (See [18,21,16] for precise statements of the assumption.)

Phrased another way, the proofs of security say the following. Suppose that our family of functions is not collision resistant; this means that there is an algorithm that, given a *randomly-chosen* function f from our family, is able to find a collision in f in some feasible amount of time T. The algorithm might only succeed on a small (but noticeable) fraction of f from the family, and may only find a collision with some small (but noticeable) probability. Given such an algorithm, there is also an algorithm that can *always* find a short nonzero vector in *any* ideal lattice over the ring $\mathbb{Z}[\alpha]/(\alpha^n + 1)$, in some feasible amount of time related to T and the success probability of the collision-finder. We stress that the best known algorithms for finding short nonzero vectors in ideal lattices require *exponential* time in the dimension n, in the worst case.

The importance of *worst-case* assumptions in lattice-based cryptography cannot be overstated. Robust cryptography requires hardness *on the average*, i.e., almost every instance of the primitive must be hard for an adversary to break. However, many lattice problems are heuristically *easy* to solve on “many” or “most” instances, but still appear hard in the worst case on certain “rare” instances. Therefore, worst-case security provides a very strong and meaningful guarantee, whereas ad-hoc assumptions on the average-case difficulty of lattice problems may be unjustified. Indeed, we are able to find collisions in compression function of the related LASH-x family of hash functions [3] by falsifying its underlying (ad-hoc) average-case lattice assumption (see Section 5.3).

At a minimum, our asymptotic proofs of security indicate that there are no unexpected “structural weaknesses” in the design of SWIFFT. Specifically, violating the claimed security properties (in an asymptotic sense) would necessarily require new algorithmic insights about finding short vectors in *arbitrary* ideal lattices (over the ring $\mathbb{Z}[\alpha]/(\alpha^n + 1)$). In Section 5.3, we demonstrate the significance of our proofs by giving examples of two compression functions from the literature that look remarkably similar to ours, but which admit a variety of very easily-found collisions.

Connection to algebraic number theory. Ideal lattices are well-studied objects from a branch of mathematics called *algebraic number theory*, the study of number fields. Let n be a power of 2, and let $\zeta_{2n} \in \mathbb{C}$ be a primitive $2n$th root of unity over the complex numbers (i.e., a root of the polynomial $\alpha^n + 1$). Then the ring $\mathbb{Z}[\alpha]/(\alpha^n + 1)$ is isomorphic to $\mathbb{Z}[\zeta_{2n}]$, which is the *ring of integers* of the so-called *cyclotomic* number field $\mathbb{Q}(\zeta_{2n})$. Ideals in this ring of integers (more generally, in the ring of integers of any number field) map to n-dimensional lattices under what is known as the *canonical embedding* of the number field. These are exactly the ideal lattices for which we assume finding short vectors is difficult in the worst case.[2] Further connections between the complexity of lattice problems and algebraic number theory were given by Peikert and Rosen [22].

For the cryptographic security of our hash functions, it is important that the extra ring structure does not make it easier to find short vectors in ideal lattices. As far as we know, and despite being a known open question in algebraic number theory, there is no apparent way to exploit this algebraic structure. The best known algorithms for finding short vectors in ideal lattices are the same as those for general lattices, and have similar performance. It therefore seems reasonable to conjecture that finding short vectors in ideal lattices is infeasible (in the worst case) as the dimension n increases.

5.2 Known Attacks

We caution that our asymptotic proofs do not necessarily rule out cryptanalysis of specific parameter choices, or ad-hoc analysis of one *fixed* function from the family. To quantify the exact security of our functions, it is still crucially important to cryptanalyze our specific parameter choices and particular instances of the function.

A central question in measuring the security of our functions is the meaning of "infeasible" in various attacks (e.g., collision-finding attacks). Even though our functions have an output length of about $n \lg p$ bits, we do *not* claim that they enjoy a full $2^{n \lg p}$ "level of security" for one-wayness, nor a $2^{(n \lg p)/2}$ level of security for collision resistance. Instead, we will estimate concrete levels of security for our specific parameter settings. This is akin to security estimates for public-key primitives like RSA, where due to subexponential-time factoring algorithms, a 1024-bit modulus may offer only (say) a 2^{100} concrete level of security.

In Section 5.2, we describe the known algorithms to find collisions in our functions takes time at least 2^{106} and requires almost as much space. We also describe the known inversion attacks, which require about 2^{128} time and space.

Throughout this section, it will be most convenient to cryptanalyze our functions using their algebraic characterization as described in Section 2.1, and in particular, Equation (1).

Connection to Subset Sum. A very useful view of our compression function is as a subset sum function in which the weights come from the additive group $\mathbb{Z}_p^n$, and are related algebraically.

An element $\mathbf{a}$ in the ring $R = \mathbb{Z}_p[\alpha]/(\alpha^n + 1)$ can be written as $a_0 + a_1\alpha + \ldots + a_{n-1}\alpha^{n-1}$, which we can represent as a vector $(a_0, \ldots, a_{n-1}) \in \mathbb{Z}_p^n$. Because $\alpha^n \equiv -1$ in the ring R, the product of two polynomials $\mathbf{a}, \mathbf{x} \in R$ is represented

[2] In [18,21,16], the mapping from ideals to lattices is slightly different, involving the coefficient vectors of elements in $\mathbb{Z}[\zeta_{2n}]$ rather than the canonical embedding. However, both mappings are essentially the same in terms of lengths of vectors, and the complexity of finding short vectors is the same under both mappings.

by the matrix product of the square *skew-circulant* matrix of $\mathbf{a}$ with the vector representation of $\mathbf{x}$:

$$\mathbf{a} \cdot \mathbf{x} \in R \quad \leftrightarrow \quad \begin{bmatrix} a_0 & -a_{n-1} & \cdots & -a_1 \\ a_1 & a_0 & \cdots & -a_2 \\ \vdots & & \ddots & \\ a_{n-1} & a_{n-2} & \cdots & a_0 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{bmatrix} \bmod p \qquad (3)$$

Thus we can interpret Equation (1) (with fixed multipliers $\mathbf{a}_1, \ldots, \mathbf{a}_m$) as multiplying a fixed matrix $\mathbf{A} \in \mathbb{Z}_p^{n \times mn}$ by an input vector $\mathbf{x} \in \{0,1\}^{mn}$. The matrix $\mathbf{A}$ has the form

$$\mathbf{A} = [\mathbf{A}_1 | \cdots | \mathbf{A}_m] \qquad (4)$$

where each $\mathbf{A}_i$ is the $n \times n$ skew-circulant matrix of $\mathbf{a}_i$. Ignoring for a moment the algebraic dependencies within each $\mathbf{A}_i$, this formulation is equivalent to a subset sum function over the group $\mathbb{Z}_p^n$. Indeed, the output of our function is just the sum of a subset of the mn column vectors of $\mathbf{A}$. And in fact, the fastest known algorithm for inverting (or finding collisions in) our function f is the same one that is used for solving the high density subset sum problem [28,15]. We describe this algorithm next.

Generalized Birthday Attack. Finding a collision in our function is equivalent to finding a nonzero $\mathbf{x} \in \{-1,0,1\}^{mn}$ such that

$$\mathbf{A}\mathbf{x} = \mathbf{0} \bmod p \qquad (5)$$

where $\mathbf{A}$ is as in Equation (4). This is because if we find a $\{-1,0,1\}$-combination of the columns of $\mathbf{A}$ that sums to $\mathbf{0} \bmod p$, the subset of the columns corresponding to the -1s collides with the subset corresponding to the 1s. We will now describe an algorithm for finding such an $\mathbf{x}$ for the specific parameters $n = 64$, $m = 16$, $p = 257$. Our goal is to provide a *lower bound* on the running time of the most efficient known algorithm for breaking our function. Therefore the analysis of the function will be fairly conservative.

Given a 64×1024 matrix $\mathbf{A}$ whose coefficients are in $\mathbb{Z}_{257}$, we proceed as follows:

1. Arbitrarily break up the 1024 column vectors of $\mathbf{A}$ into 16 groups of 64 vectors each.
2. From each group, create a list of 3^{64} vectors where each vector in the list is a different $\{-1,0,1\}$-combination of the vectors in the group.

We now have 16 lists each containing $3^{64} \approx 2^{102}$ vectors in $\mathbb{Z}_{257}^{64}$. Notice that if we are able to find one vector from each list such that their sum is the zero vector, then we can solve Equation 5.

Finding one vector from each list such that the sum is $\mathbf{0}$ is essentially the k-list problem that was studied by Wagner [28], and is also related to the technique used by Blum, Kalai, and Wassserman [5] for solving the parity problem in the presence of noise. The idea is to use the lists to obtain new lists of vectors that are $\{-1,0,1\}$-combinations of $\mathbf{A}$'s columns, but which have many coordinates that are 0. We then continue forming lists in which the vectors have more and more coordinates equal to 0. More precisely, we continue with the algorithm in the following way:

3. Pair up the 16 lists in an arbitrary way.
4. For each pair of lists (L_i, L_j), create a new list $L_{i,j}$ such that every vector in $L_{i,j}$ is the sum of one vector from L_i and one vector from L_j, and the first 13 positions of the vector are all 0 modulo 257.

There are a total of $257^{13} \approx 2^{104}$ different values that a vector in $\mathbb{Z}_{257}^{64}$ can take in its first 13 entries. Since the lists L_i and L_j each contain $3^{64} \approx 2^{102}$ vectors, there are a total of 2^{204} possible vectors that could be in $L_{i,j}$. If we heuristically assume that each of the $257^{13} \approx 2^{104}$ possible values of the first 13 coordinates are equally likely to occur[3], then we expect the list $L_{i,j}$ to consist of $2^{204} \cdot 2^{-104} = 2^{100}$ vectors whose first 13 coordinates are all 0. For convenience, we will assume that the lists have 2^{102} vectors (this again is a conservative assumption that is in the algorithm's favor).

At the end of Step 4, we have 8 lists, each with 2^{102} vectors in $\mathbb{Z}_{257}^{64}$ whose first 13 coordinates are zero. We can now pair up these 8 lists and create 4 lists of 2^{102} vectors whose first 26 coordinates are zero. We continue until we end up with one list of 2^{102} elements whose first 52 coordinates are zero. This means that only the last 12 coordinates of these vectors may be nonzero. If the vectors are randomly distributed in the last 12 coordinates, then there should be a vector which consists of all zeros (because there are only $257^{12} \approx 2^{96}$ possibilities for the last 12 coordinates).

Since we started out with 16 lists of 2^{102} elements, the running time of the algorithm is at least $16 \cdot 2^{102} = 2^{106}$. Notice that it also requires at least 2^{102} space.

Using a slightly modified generalized birthday attack, it is also possible to mount an inversion attack using time and space approximately 2^{128}. The main difference is that we use 8 lists, and when combining (say) the first two lists in each level of the tree, we pair up entries so that they match the desired output value on the appropriate block of entries (for the other pairs of lists, we pair the entries to produce zeros).

Lattice Attacks. Lattice reduction is a possible alternative way to find a nonzero vector $\mathbf{x} \in \{-1,0,1\}^{mn}$ that will satisfy Equation (5). If we think of the matrix $\mathbf{A}$ as defining a linear homomorphism from $\mathbb{Z}^{mn}$ to $\mathbb{Z}_p^n$, then the kernel of $\mathbf{A}$ is $\ker(\mathbf{A}) = \{\mathbf{y} \in \mathbb{Z}^{mn} : \mathbf{A}\mathbf{y} \equiv \mathbf{0} \bmod p\}$. Notice that $\ker(\mathbf{A})$ is a lattice of dimension mn, and a vector $\mathbf{x} \in \{-1,0,1\}^{mn}$ such that $\mathbf{A}\mathbf{x} \equiv \mathbf{0} \bmod p$ is a shortest nonzero vector in the ℓ_∞ or "max" norm of this lattice.

Because a basis for $\ker(\mathbf{A})$ can be computed efficiently given $\mathbf{A}$, finding a shortest nonzero vector (in the ℓ_∞ norm) of the lattice would yield a collision

[3] This is true if all the vectors in the lists are random and independent in $\mathbb{Z}_{257}^{64}$, but this is not quite the case. Nevertheless, since we are being conservative, we will assume that the algorithm will still work.

in our compression function. The lattice $\ker(\mathbf{A})$ shares many properties with the commonly occurring knapsack-type lattice (see, e.g., [20]). Our lattice is essentially a knapsack-type lattice with some additional algebraic structure. It is worthwhile to note that none of the well-known lattice reduction algorithms take advantage of the algebraic structure that arises here. Because the dimension 1024 of our lattice is too large for the current state-of-the-art reduction algorithms, breaking our function via lattice reduction would require some very novel idea to exploit the additional algebraic structure. As things stand right now, we believe that the generalized birthday technique described in the previous section provides a more efficient algorithm for finding collisions in our function.

Viewing the kernel as a lattice also leads naturally to a relaxed notion of "pseudo-collisions" in our function, which are defined in terms other norms, e.g., the Euclidean ℓ_2 norm or "Manhattan" ℓ_1 norm. Note that for any *actual* collision corresponding to an $\mathbf{x} \in \ker(\mathbf{A})$, we have $\mathbf{x} \in \{-1, 0, 1\}^{mn}$ and therefore the ℓ_2 norm of $\mathbf{x}$ is $\|\mathbf{x}\|_2 \leq \sqrt{mn}$. However, not every nonzero $\mathbf{x} \in \ker(\mathbf{A})$ with $\|\mathbf{x}\|_2 \leq \sqrt{mn}$ determines a collision in our function, because the entries of $\mathbf{x}$ may lie outside $\{-1, 0, 1\}$. We say that such an $\mathbf{x}$ is a *pseudo-collision* for the ℓ_2 norm. More generally, a pseudo-collision for any ℓ_p norm ($1 \leq p < \infty$) is defined to be a nonzero $\mathbf{x} \in \ker(\mathbf{A})$ such that $\|\mathbf{x}\|_p \leq (mn)^{1/p}$. The set of pseudo-collisions only grows as p decreases from ∞ to 1, so finding pseudo-collisions using lattice reduction might be easier in norms such as ℓ_2 or ℓ_1. Finding pseudo-collisions could be a useful starting point for finding true collisions.

5.3 Cryptanalysis of Similar Functions

In this section, we briefly discuss two other compression functions appearing in the literature that bear a strong resemblance to ours, but which *do not* have asymptotic proofs of collision resistance. In fact, these compression functions are *not* collision resistant, and admit quite simple collision-finding algorithms. The attacks are made possible by a structural weakness that stems from the use of *circulant* matrices, which correspond algebraically to rings that are *not integral domains*. Interestingly, integral domains are the crucial ingredient in the asymptotic proofs of collision resistance for our function [21,16]. We believe that this distinction underscores the usefulness and importance of security proofs, especially *worst-case* hardness proofs for lattice-based schemes.

Micciancio's Cyclic One-Way Function. The provably *one-way* function described by Micciancio [18] is very similar to SWIFFT, and was the foundation for the subsequent collision-resistant functions on which this paper is based [21,16]. Essentially, the main difference between Micciancio's function and the ones presented in [21,16] is that the operations are performed over the ring $\mathbb{Z}_p[\alpha]/(\alpha^n - 1)$ rather than $\mathbb{Z}_p[\alpha]/(\alpha^n + 1)$. This difference, while seemingly minor, makes it almost trivial to find collisions, as shown in [21,16].

Just like ours, Micciancio's function has an interpretation as the product of a matrix $\mathbf{A}$ (as in Equation (4)) and a vector $\mathbf{x} \in \{0, 1\}^{mn}$. The only difference is the matrices $\mathbf{A}_i$ from Equation (4) are *circulant*, rather than skew-circulant (i.e., just like Equation (3), but without negations).

Notice that in the vector product of a circulant matrix $\mathbf{A}_i$ with the all-1s vector $\mathbf{1}$, all of the entries are the same. Thus, for any circulant matrix $\mathbf{A}_i$, the n-dimensional vector $\mathbf{A}_i \cdot \mathbf{1} \bmod p$ can be only one of p distinct vectors. There are 2^m ways to set each vector $\mathbf{x}_1, \ldots, \mathbf{x}_m$ to be either $\mathbf{0}$ or $\mathbf{1}$, but there are only p distinct values of the compression function $\mathbf{Ax} = \mathbf{A}_1\mathbf{x}_1 + \ldots + \mathbf{A}_m\mathbf{x}_m \bmod p$. Because $2^m > p$ (otherwise the function does not compress), some pair of distinct binary vectors are mapped to the same output. Such a collision can be found in linear time.

LASH Compression Function. LASH-x is a family of hash functions that was presented at the second NIST hash function workshop [3]. Its compression function $f_{\mathbf{H}}$ takes an n-bit input $\mathbf{x} = \mathbf{x}_1|\mathbf{x}_2$ where $\mathbf{x}_1, \mathbf{x}_2 \in \{0,1\}^{n/2}$, and is defined as $f_{\mathbf{H}}(\mathbf{x}) = (\mathbf{x}_1 \oplus \mathbf{x}_2) + \mathbf{Hx} \bmod q$, where $\mathbf{H}$ is a "semi-circulant" $m \times n$ matrix whose entries are from the group $\mathbb{Z}_{256}$:

$$\mathbf{H} = \begin{bmatrix} a_0 & a_{n-1} & a_{n-2} & \cdots & a_1 \\ a_1 & a_0 & a_{n-1} & & a_2 \\ \vdots & & & \ddots & \\ a_{m-1} & a_{m-2} & a_{m-3} & \ldots & a_m \end{bmatrix}.$$

The values $a_0, \ldots, a_{n-1} \in \mathbb{Z}_{256}$ are essentially chosen at random (actually, according to a weak pseudorandom generator).

As discussed in [3], a heuristic assumption for the security of LASH is that its compression function $f_{\mathbf{H}}$ is collision-resistant. However, for a random choice of the entries $a_0, \ldots, a_{n-1}$, finding a collision in $f_{\mathbf{H}}$ with noticeable probability is actually trivial. Notice that all the rows of the matrix $\mathbf{H}$ have the same sum. If this sum happens to be 0 mod 256 (which happens with probability 1/256 over the choice of the a_i), then we have

$$f(\mathbf{0}) = 0 + \mathbf{H} \cdot \mathbf{0} = \mathbf{0} = 0 + \mathbf{H} \cdot \mathbf{1} = f(\mathbf{1}),$$

where $\mathbf{0}$ and $\mathbf{1}$ are all-0s and all-1s vectors, respectively. Therefore these two distinct inputs make up a collision in the compression function.

When n is divisible by a large power of 2 (e.g., $n = 640, 1024$ are proposed in [3]), other collisions may be easy to find as well (with some noticeable probability over the choice of the a_i). For example, the inputs $\mathbf{x} = 0101 \cdots 01$ and $\mathbf{x}' = 1010 \cdots 10$ will collide with probability 1/256, because $\mathbf{Hx}$ and $\mathbf{Hx}'$ consist of two repeated values, i.e., the sum of the even-indexed a_is and the sum of the odd-indexed a_is. Other kinds of collisions are also possible, corresponding essentially to the factorization of the polynomial $\alpha^n - 1$ over $\mathbb{Z}[\alpha]$.

The attacks described above apply only to LASH's underlying compression function, and not (as far as we are aware) to the full LASH hash function itself. Using different ideas (that do not exploit the above-described structural weakness in $f_{\mathbf{H}}$), Contini *et al* [8] give a thorough cryptanalysis of the full LASH hash function.

Acknowledgments

We thank Ron Steinfeld and Gaëtan Leurent (among others) for pointing out the more efficient inversion attacks using the generalized birthday attack.

References

1. Ajtai, M.: Generating hard instances of lattice problems. In: STOC, pp. 99–108 (1996)
2. Baritaud, T., Gilbert, H., Girault, M.: FFT hashing is not collision-free. In: Rueppel, R.A. (ed.) EUROCRYPT 1992. LNCS, vol. 658, pp. 35–44. Springer, Heidelberg (1993)
3. Bentahar, K., Page, D., Silverman, J., Saarinen, M., Smart, N.: Lash. Technical report, 2nd NIST Cryptographic Hash Function Workshop (2006)
4. Biham, E., Chen, R., Joux, A., Carribault, P., Jalby, W., Lemuet, C.: Collisions of SHA-0 and reduced SHA-1. In: Cramer, R.J.F. (ed.) EUROCRYPT 2005. LNCS, vol. 3494. Springer, Heidelberg (2005)
5. Blum, A., Kalai, A., Wasserman, H.: Noise-tolerant learning, the parity problem, and the statistical query model. Journal of the ACM 50(4), 506–519 (2003)
6. Cai, J., Nerurkar, A.: An improved worst-case to average-case connection for lattice problems. In: FOCS, pp. 468–477 (1997)
7. Camion, P., Patarin, J.: The knapsack hash function proposed at Crypto 1989 can be broken. In: Quisquater, J.-J., Vandewalle, J. (eds.) EUROCRYPT 1989. LNCS, vol. 434, pp. 39–53. Springer, Heidelberg (1990)
8. Contini, S., Matusiewicz, K., Pieprzyk, J., Steinfeld, R., Guo, J., Ling, S., Wang, H.: Cryptanalysis of LASH. Cryptology ePrint Archive, Report 2007/430 (2007), http://eprint.iacr.org/
9. Daemen, J., Bosselaers, A., Govaerts, R., Vandewalle, J.: Collisions for Schnorr's hash function FFT-hash presented at crypto 1991. In: Matsumoto, T., Imai, H., Rivest, R.L. (eds.) ASIACRYPT 1991. LNCS, vol. 739. Springer, Heidelberg (1993)
10. Damgård, I.: A design principle for hash functions. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 416–427. Springer, Heidelberg (1990)
11. Goldreich, O., Goldwasser, S., Halevi, S.: Collision-free hashing from lattice problems. Technical Report TR-42, ECCC (1996)
12. Goldreich, O., Goldwasser, S., Micali, S.: How to construct random functions. J. ACM 33(4), 792–807 (1986)
13. Hoffstein, J., Pipher, J., Silverman, J.H.: NTRU: A ring-based public key cryptosystem. In: ANTS, pp. 267–288 (1998)
14. Joux, A., Granboulan, L.: A practical attack against knapsack based hash functions (extended abstract). In: De Santis, A. (ed.) EUROCRYPT 1994. LNCS, vol. 950, pp. 58–66. Springer, Heidelberg (1995)
15. Lyubashevsky, V.: The parity problem in the presence of noise, decoding random linear codes, and the subset sum problem. In: Chekuri, C., Jansen, K., Rolim, J.D.P., Trevisan, L. (eds.) APPROX 2005 and RANDOM 2005. LNCS, vol. 3624, pp. 378–389. Springer, Heidelberg (2005)
16. Lyubashevsky, V., Micciancio, D.: Generalized compact knapsacks are collision resistant. In: Bugliesi, M., Preneel, B., Sassone, V., Wegener, I. (eds.) ICALP 2006. LNCS, vol. 4052, pp. 144–155. Springer, Heidelberg (2006)

17. Micciancio, D.: Almost perfect lattices, the covering radius problem, and applications to Ajtai's connection factor. SIAM J. on Computing 34(1), 118–169 (2004)
18. Micciancio, D.: Generalized compact knapsacks, cyclic lattices, and efficient one-way functions from worst-case complexity assumptions. Computational Complexity 16, 365–411 (2007); Preliminary version in FOCS 2002
19. Micciancio, D., Regev, O.: Worst-case to average-case reductions based on Gaussian measures. SIAM J. on Computing 37(1), 267–302 (2007)
20. Nguyen, P., Stehlé, D.: LLL on the average. In: ANTS, pp. 238–256 (2006)
21. Peikert, C., Rosen, A.: Efficient collision-resistant hashing from worst-case assumptions on cyclic lattices. In: Halevi, S., Rabin, T. (eds.) TCC 2006. LNCS, vol. 3876. Springer, Heidelberg (2006)
22. Peikert, C., Rosen, A.: Lattices that admit logarithmic worst-case to average-case connection factors. In: STOC, pp. 478–487; Full version in ECCC Report TR06-147 (2007)
23. Rogaway, P., Shrimpton, T.: Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In: Roy, B., Meier, W. (eds.) FSE 2004. LNCS, vol. 3017, pp. 371–388. Springer, Heidelberg (2004)
24. Schnorr, C.P.: FFT-hash, an efficient cryptographic hash function. In: Crypto Rump Session (1991)
25. Schnorr, C.P.: FFT–Hash II, efficient cryptographic hashing. In: Rueppel, R.A. (ed.) EUROCRYPT 1992. LNCS, vol. 658, pp. 45–54. Springer, Heidelberg (1993)
26. Schnorr, C.P.: Serge Vaudenay. Parallel FFT-hashing. In: Fast Software Encryption, pp. 149–156 (1993)
27. Vaudenay, S.: FFT-Hash-II is not yet collision-free. In: Brickell, E.F. (ed.) CRYPTO 1992. LNCS, vol. 740, pp. 587–593. Springer, Heidelberg (1993)
28. Wagner, D.: A generalized birthday problem. In: Yung, M. (ed.) CRYPTO 2002. LNCS, vol. 2442, pp. 288–303. Springer, Heidelberg (2002)
29. Wang, X., Lai, X., Feng, D., Chen, H., Yu, X.: Cryptanalysis for hash functions MD4 and RIPEMD. In: Cramer, R.J.F. (ed.) EUROCRYPT 2005. LNCS, vol. 3494. Springer, Heidelberg (2005)
30. Wang, X., Yu, H.: How to break MD5 and other hash functions. In: Cramer, R.J.F. (ed.) EUROCRYPT 2005. LNCS, vol. 3494. Springer, Heidelberg (2005)

A Unified Approach to Related-Key Attacks

Eli Biham[1,⋆], Orr Dunkelman[2,⋆⋆], and Nathan Keller[3,⋆⋆⋆]

[1]Computer Science Department, Technion
Haifa 32000, Israel
biham@cs.technion.ac.il
[2]ESAT/SCD-COSIC, Katholieke Universiteit Leuven
Kasteelpark Arenberg 10, B-3001 Leuven-Heverlee, Belgium
orr.dunkelman@esat.kuleuven.be
[3]Einstein Institute of Mathematics, Hebrew University
Jerusalem 91904, Israel
nkeller@math.huji.ac.il

Abstract. This paper introduces a new framework and a generalization of the various flavors of related-key attacks. The new framework allows for combining all the previous related-key attacks into a complex, but much more powerful attack. The new attack is independent of the number of rounds of the cipher. This property holds even when the round functions of the cipher use different subkeys.

The strength of our new method is demonstrated by an attack on $4r$-round IDEA, for any r. This attack is the first attack on a widely deployed block cipher which is independent of the number of rounds. The variant of the attack with $r = 2$ is the first known attack on 8-round IDEA.

1 Introduction

In many applications the same block cipher is used with two unknown keys whose relation is known. To study the security of block ciphers in these situations the *Related-key attacks* framework was first presented in 1993 [3]. In a related-key attack, the attacker is allowed to ask for plaintexts encrypted under two (or more) related keys. This approach might seem unrealistic, as it assumes that the attacker can control some relations between the unknown keys. Still, there are some instances, e.g., the 2PKDP protocol [43], where this approach suggests practical attacks.

A block cipher susceptible to a related-key attack has some security concerns. It may not be suitable for other cryptographic primitives that use block ciphers

[⋆] This work was supported in part by the Israel MOD Research and Technology Unit.

[⋆⋆] This work was supported in part by the Concerted Research Action (GOA) Ambiorics 2005/11 of the Flemish Government and by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy).

[⋆⋆⋆] The research presented in this paper was supported by the Adams fellowship.

K. Nyberg (Ed.): FSE 2008, LNCS 5086, pp. 73–96, 2008.

as building blocks, e.g., hash functions. A famous example for this claim is the block cipher TEA [45]. A related-key property of TEA [33] was used in hacking Microsoft's Xbox architecture, which uses a Davies-Meyer hash function employing TEA as the underlying block cipher [46]. Another security concern is the fact that such a cipher cannot be used in protocols which allow key manipulation, such as the ones used in most inter-bank communications in the US which increment the key by one in each transaction. Sometimes, the security of the mode of operation of the block cipher is closely related to the immunity of the cipher to related-key attacks (as in the 3GPP case, as was shown in [29]).

There are two classes of related-key attacks: The first class, originally presented by Biham [3] and independently by Knudsen [36], are attacks that use related-key plaintext pairs. These attacks use pairs of keys for which most of the encryption function is equivalent. Such relations exist when the key schedule is very simple. Also, in order for the attacks to succeed, the round functions have to be relatively weak (i.e., there exists a known plaintext attack on the round function given one or two input/output pairs). On the other hand, once such a relation can be found, it can be used to devise an attack on the cipher, where the attack is independent of the number of rounds.

The second class of related-key attacks, originally presented by Kelsey et al. [32,33] is composed of attacks that treat the key relation as another freedom level in the examination of statistical properties of the cipher. Besides related-key differentials, where the key difference is used to control the evolution of differences, this class contains variants of most of the known cryptanalytic techniques: The SQUARE attack [20] was treated in the related-key model in [23] and used to extend the best known SQUARE attack against AES into a related-key attack that uses 256 related keys. The boomerang attack [44] and the rectangle attack [5] were combined with related-key differentials to introduce the related-key boomerang and related-key rectangle attacks [7,28,35]. Finally, linear cryptanalysis [38] was also combined with related-key attacks to produce a related-key attack on 7.5-round IDEA [8]. The second class of attacks can deal with much more complex key schedules and round functions, but their effectiveness (usually) drops with the number of rounds.

In this paper we unify the main ideas from the two classes of related-key attacks into one framework. The new framework has two main advantages:

1. A new approach for generating multiple related-key plaintext pairs, based on multiple encryptions under chains of related keys. When the key schedule has a short cycle, it is possible to obtain many related-key plaintext pairs from one pair using encryption under several keys. This technique allows to mount attacks on round functions that require more than two input/output pairs.
2. A combination of the two classes of attacks to allow a related-key attack on the underlying round function (rather than applying only a simple attack on the round function).

Thus, the unified approach allows attacking more ciphers, as the restrictions on the key schedule and on the round functions are significantly reduced.

To demonstrate the strength of the new technique, we apply our new attack to $4r$-round IDEA [37]. IDEA is a 64-bit block cipher with 128-bit keys, which was introduced by Lai and Massey in 1991. IDEA was thoroughly analyzed [1,4,7,8,9,12,16,18,19,21,22,26,27,31,39,40,41] but the best known attack on the cipher is against 7.5-round IDEA (out of 8.5 rounds) in the related-key model [8], and 6-round IDEA in the single key model [9].

We first introduce a related-key attack on 4-round IDEA. Then, using our new framework, we elevate this attack to any $4r$-round IDEA, presenting the first attack on IDEA that is independent of the number of rounds. The variant of the attack with $r = 2$ is the first known attack on 8-round IDEA.

The remainder of this paper is organized as follows: In Section 2 we present the various related-key attacks. We incorporate all the attacks into the new related-key framework in Section 3. Section 4 describes our attack on $4r$-round IDEA. Appendix A contains a short description of IDEA. Appendix B gives the full description of the 4-round related-key differential of IDEA we use in our attack. In Appendix C we outline a different attack algorithm on $4r$-round IDEA (with roughly the same data and time complexities). We conclude the paper in Section 5.

2 Previous Work

Related-key attacks exploit the relations between the encryption processes under different but related keys. Related-key attacks can be divided into two classes. The first class is attacks concentrated on detecting and exploiting related-key plaintext pairs, and the second class is adaptation of the standard cryptanalytic attacks into the related-key model.

2.1 Related-Key Attacks Exploiting Related-Key Plaintext Pairs

The original variant of related-key attacks introduced in [3,36] is attacks exploiting related-key plaintext pairs. The main idea behind the attack is to find instances of keys for which the encryption processes deploy the same permutation (or almost the same permutation). To illustrate the technique, we shortly present the attack from [3].

Consider a variant of DES in which all the rotate left operations in the key schedule algorithm are by a fixed number of bits.[1] For any key K^1 there exists another key K^2 such that the round subkeys KR^1_i, KR^2_i produced by K^1 and K^2, respectively, satisfy:

$$KR^1_{i+1} = KR^2_i, \qquad \text{for } i = 1, \ldots, 15.$$

For such pair of keys, if a pair of plaintexts (P_1, P_2) satisfies $P_2 = f_{KR^1_1}(P_1)$, where $f_{sk}(P)$ denotes one round DES encryption of P under the subkey sk, then the corresponding ciphertexts C_1 and C_2 satisfy $C_2 = f_{KR^2_{16}}(C_1)$. Given such a

[1] Such a variant was proposed by Brown and Seberry [17].

pair of plaintexts (called in the sequel *a related-key plaintext pair*), the subkeys KR_{16}^2 and KR_1^1 can be easily extracted [3].

Throughout the paper we shall refer to the round in which the key is recovered as the *underlying round function.* This kind of related-key attacks is based on finding related-key plaintext pairs, which are then used to extract an input/output pair (or two pairs) to the round function. Once the input/output pair to the underlying round function is given, the attacker applies a cryptanalytic attack on the underlying round function and retrieves the key.

In the above attack, the attacker asks for the encryption of two pools of 2^{16} chosen plaintexts under two (unknown) related keys K^1, K^2. The plaintexts in the first pool, denoted by S_1, are of the form (X, A) and are encrypted under K^1, and the plaintexts in the second pool, denoted by S_2, are of the form (A, Y) and are encrypted under K^2, where A is some fixed 32-bit value and X, Y vary.

The attacker then finds pairs of ciphertexts (C_1, C_2), such that the first ciphertext belongs to S_1 and the second one belongs to S_2 and such that the left half of C_1 equals to the right half of C_2. Once such a pair is found, then there is a good chance that P_1 and P_2, the corresponding plaintexts, satisfy that $P_2 = f_{KR_1^1}(P_1)$. If this is the case, then the pair (P_1, P_2) is a related-key plaintext pair, and it can be used to retrieve the values of KR_{16}^2 and KR_1^1. It can be shown that with a high probability, if the pair is not a related-key plaintext pair, this procedure yields a contradiction, and hence, once a consistent value for KR_{16}^2 and KR_1^1 is suggested by the attack, it is the correct value with high probability.

The data complexity of the attack is 2^{17} chosen plaintexts, and the time complexity of the attack is 2^{17} encryptions as well. We note that even if there were more rounds in the modified version of DES, the attack would still be successful.

In the more general case, this class of related-key attacks is composed of three parts: Obtaining related-key plaintexts, identifying the related-key plaintext pairs, and using them to deduce the key. In many cases, identifying the related-key plaintext pairs is best achieved by assuming for each candidate pair that it is a related-key plaintext pair, and then using it as an input for the key recovery phase of the attack. In other cases, the round functions' weaknesses allow the attacker to identify these pairs easily.

The attack relies heavily on the simplicity of the key schedule, the similarity of the rounds, and on the cryptographic weakness of the underlying round function. As a result, most of the known block ciphers are immune to related-key attacks of this class.

2.2 Slide Attacks

When a cipher has self-related keys, i.e., it can be written as $E_k = f_k^\ell = f_k \circ f_k \circ \cdots \circ f_k$ it is susceptible to a variant of the related-key attack called the *slide attack* [13]. In this case, it is possible to apply the related-key attack to the cipher with $K^1 = K^2$, thus eliminating the key requirement of having two keys. The attacker looks for a slid pair, i.e., two plaintexts (P_1, P_2) such that

$P_2 = f_k(P_1)$. In this case, the pair satisfies $C_2 = f_k(C_1)$ as well. When the round function f_k is simple enough, it is possible to use these two pairs in order to deduce information about the key.

In the slide attack the attacker obtains enough plaintext/ciphertext pairs to contain a slid pair, and has to check for each possible pair of plaintexts whether it is a slid pair by applying the attack on f_k. When dealing with a general block cipher this approach requires $O(2^{n/2})$ known plaintexts and $O(2^n)$ applications of the attack on f_k, where n is the block size. For Feistel block ciphers, the attack can be optimized using $O(2^{n/4})$ chosen plaintexts and $O(1)$ applications of the attack. Note that as in the original related-key attacks, the main drawback of this approach is that the attack can be used only if f_k can be broken using only two known input/output pairs, i.e., given one slid pair.

In 2000, Biryukov and Wagner [14] presented two variants of the slide attack, named *complementation slide* and *sliding with a twist*. These variants allow for treating more complex functions in the slide attack. Nevertheless, there are no widely used ciphers that can be attacked using these techniques.

The authors of [14] also presented several techniques aimed at finding several slid pairs simultaneously, enabling to use the attack even if several input/output pairs are needed for attacking f_k. One of these techniques, fully explored by Furuya [24], uses the fact that when (P_1, P_2) is a slid pair then $(E_k^t(P_1), E_k^t(P_2))$ are also slid pairs for all values of t.[2] This allows the attacker to transform any known plaintext attack on f_k that requires m known plaintexts to an attack on E_k with a data complexity of $O(m \cdot 2^{n/2})$ adaptively chosen plaintexts. The time complexity of this approach is $O(2^n)$ applications of the known plaintext attack on f_k.[3]

2.3 Attacks Adapting Standard Techniques to the Related-Key Model

The related-key model can be used as a platform for all standard attacks. This fact was first noted in [32,33] where related-key differentials were introduced. Recall, that a regular differential deals with some plaintext difference ΔP and a ciphertext difference ΔC such that

$$\Pr{}_{P,K}[E_K(P) \oplus E_K(P \oplus \Delta P) = \Delta C]$$

is high enough. A related-key differential is a triplet of a plaintext difference ΔP, a ciphertext difference ΔC, and a key difference ΔK, such that

$$\Pr{}_{P,K}[E_K(P) \oplus E_{K \oplus \Delta K}(P \oplus \Delta P) = \Delta C]$$

is high enough.

[2] Throughout the paper the notation $F^i(\cdot)$ means i successive applications of $F(\cdot)$.

[3] It is worth mentioning that the technique can be easily improved in the case of Feistel ciphers, for which $O(2^{n/4})$ chosen plaintexts and $O(m)$ adaptive chosen plaintexts are sufficient to achieve m slid pairs, which are easily identified.

The related-key differential attack uses the subkey differences to control the development of differences through the encryption process. As a result, related-key differentials are usually much stronger than the respective "ordinary" differentials. The related-key differential technique was used to attack numerous block ciphers, including GOST, TEA, and 6-round KASUMI.

For example, using this approach a 60-round related-key differential with probability 2^{-30} of TEA is presented. Using this related-key differential, it is possible to break the full TEA (with 64 rounds) using about 2^{32} related-key chosen plaintexts and a small computational effort [33].

After the introduction of impossible differentials, i.e., differentials with zero probability, in [4], the concept of related-key impossible differentials was presented [30]. In this case, the subkey relations are used to ensure that the input difference of the impossible differential can not evolve into the output difference. This technique was used to analyze 8-round AES-192.

Related-key differentials were also used as the base for the related-key boomerang and the related-key rectangle attacks [7,28,35]. These attacks use two related-key differentials, i.e., up to four related keys. Hence, they enjoy the transition into related-key differentials twice, leading to much higher probabilities for the distinguisher (in exchange for more related keys). The related-key rectangle technique was successfully applied to several block ciphers, including 10-round AES-192, 10-round AES-256, the full SHACAL-1, the full KASUMI, and 7-round IDEA.

In [23] it is showed that the SQUARE attack can also be improved in the related-key model. The 4-round SQUARE property of AES used in the regular SQUARE attack, is extended into a 5-round related-key SQUARE property for AES-256. As a result, while the ordinary SQUARE technique can be used to attack up to 7 rounds of AES, the related-key SQUARE attack is applicable to a 9-round variant of AES.

Finally, even linear relations can be improved in the related-key model. In [8] a 2.5-round linear relation of IDEA is presented. When two related keys are used, this linear relation can be extended to a 4.5-round linear relation. This extension improves the regular attack on IDEA by 2.5 rounds, and is the best known attack so far against IDEA.

3 The Unified Related-Key Framework

In this section we present the new framework unifying the different related-key attacks. The construction of the framework is divided into two stages:

- First, we present a new approach to generating multiple related-key plaintext pairs, based on encryption under chains of related keys. This approach allows to mount a related-key attack on the entire cipher, even if the attack on the underlying function requires multiple input/output pairs.
- Then, we unify the two classes of related-key attacks into a single framework. This allows to use a related-key attack on the underlying round function.

Thus, even if the underlying round function is secure against regular cryptanalytic attacks, it can still be attacked using a related-key attack.

3.1 A New Approach for Generating Multiple Related-Key Plaintext Pairs

When a cipher can be written as $E_k = f_k^\ell = f_k \circ f_k \circ \cdots \circ f_k$ and the slide attack can be applied, any slid pair (P, Q) can be used to generate many additional slid pairs of the form $(E_k^t(P), E_k^t(Q))$, for all t. These pairs can be used to devise a slide attack on the cipher even if multiple input/output pairs are required to break the underlying round function. However, this property exists only since the relation between the plaintexts of the slid pair is similar to the relation between the ciphertexts. If the plaintexts satisfy $Q = f_k(P)$ then the ciphertexts satisfy $E_k(Q) = f_k(E_k(P))$, and thus can be treated as the plaintexts in a new slid pair.

For an encryption with different subkeys, i.e., when $E_K = f_{k_r} \circ f_{k_{r-1}} \circ \cdots \circ f_{k_1}$, the situation is more complicated. The plaintexts of a related-key plaintext pair satisfy $Q = f_{k_1}(P)$, but the respective ciphertexts satisfy $E_k(Q) = f_{\text{``}k_{r+1}\text{''}}(E_k(P))$.[4] Hence, unless $k_1 =$ "k_{r+1}", multiple encryption does not yield additional related-key plaintext pairs.

Our new approach uses chains of keys in order to achieve the additional related-key plaintext pairs. Let $E_K = f_{k_r} \circ f_{k_{r-1}} \circ \cdots \circ f_{k_1}$, and let (P^1, Q^1) be a related-key plaintext pair (with the corresponding ciphertexts (P^2, Q^2)) with respect to the keys (K_P^1, K_Q^1), i.e., $k_{i+1}^{P_1} = k_i^{Q_1}$, and $f_{k_1^{P_1}}(P^1) = Q^1$. Then, if K_P^2 is a key such that $k_1^{P_2} = k_r^{Q_1}$, then $f_{k_1^{P_2}}(P^2) = Q^2$. Moreover, let K_Q^2 satisfy that $k_{i+1}^{P_2} = k_i^{Q_2}$, then (P^2, Q^2) is a related-key plaintext pair with respect to (K_P^2, K_Q^2).

Defining K_P^2 as a function of K_P^1 is usually very simple, and usually it is the key that produces the next r subkeys if the key schedule would have been extended by r rounds. Formally, there are cases in which there exists a function $g(\cdot)$ such that for every pair of keys $(K_P^1, K_P^2 = g(K_P^1))$, and the key K_Q^1 related to K_P^1, we have $k_1^{P_2} = k_r^{Q_1}$. For example, in the modified variant of DES considered in Section 2, g can be the rotation of the key by 16 times the rotation in each round. Examples of real ciphers for which such g exists are $4r$-IDEA and the full SHACAL-1. In IDEA, where each f_{k_i} represents 4 rounds, we have $g(K) = K \lll (75 \cdot r)$. For SHACAL-1, $g(K)$ is obtained from K by running the LFSR used in the key schedule of the cipher 80 steps forward.

Assume now that for the examined cipher there exists a function g as described above. If the pair (P^1, Q^1) is a related-key plaintext pair with respect to the keys (K_P^1, K_Q^1), then the corresponding ciphertext pair (P^2, Q^2) is a related-key plaintext pair with respect to the keys $(g(K_P^1), K_Q^2)$, where K_Q^2 is the key related to $g(K_P^1)$ (and in many cases it is $g(K_Q^1)$).

[4] "k_{r+1}" is the rth subkey produced by the second key. It can be treated as the $r+1$th subkey produced by the first key.

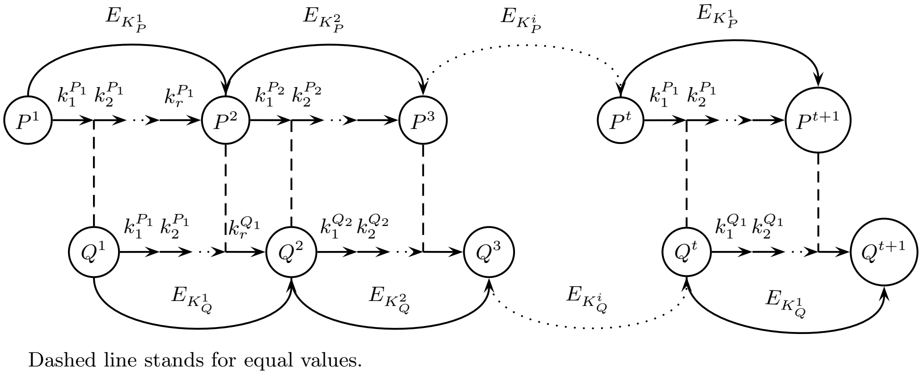


Fig. 1. The Evolution of Multiple Related-Key Plaintext Pair

This process can be repeated to achieve multiple related-key plaintext pairs with respect to different pairs of related keys. We define $H^t_{K_P} = E_{g^{t-1}(K_P)} \circ E_{g^{t-2}(K_P)} \circ \ldots \circ E_{K_P}$ and similarly $H^t_{K_Q} = E_{g^{t-1}(K_Q)} \circ E_{g^{t-2}(K_Q)} \circ \ldots \circ E_{K_Q}$. If (P^1, Q^1) is a related-key plaintext pair with respect to (K_P, K_Q), then the pair $(H^t_{K_P}(P^1), H^t_{K_Q}(Q^1))$ is a related-key plaintext with respect to $(g^t(K_P), g^t(K_Q))$.

While in some cases obtaining many related-key plaintext pairs under different keys might be useful, we have not identified a concrete example where it can be used as is. We do note that for some specific cases this property can be used to identify the related-key plaintext pairs more easily. Assume that the related-key plaintext pair satisfies some relation in the ciphertexts which is not sufficient for the immediate identification of the related-key plaintext pair (for example, $n/4$ bits out of the n bits of the ciphertexts have to be equal). It is possible to identify the related-key plaintext pair by using the fact that a related-key plaintext pair is expanded into several such ones.

In most cases though, the attack on the underlying function requires several input/output pairs encrypted under the same key. However, we note that if for some t, $g^t(K^1_P) = K^1_P$, we can get more related-key plaintext pairs under the original key pair (K^1_P, K^1_Q). We outline the evolution of such a pair in Figure 1.

For example, in $4r$-IDEA the cycle length of g for all the keys equals at most 64. Hence, using the algorithm presented above we can generate efficiently many related-key plaintext pairs encrypted under the same pair of related keys. For block ciphers whose key schedule is based on LFSRs, e.g., SHACAL-1, the cycle size is $lcm(r, l)$ where l is the cycle length of the LFSR and r is the number of rounds of the cipher.

After obtaining enough related-key plaintext pairs under the keys (K_P, K_Q) it is possible to mount any known plaintext attack on the underlying round function, similarly to the slide case. If sufficiently many known plaintexts are available, then it might be possible to mount chosen plaintext attacks, or even adaptive chosen plaintext attacks. Therefore, the new approach for the generation of related-key plaintext pairs allows to mount the related-key attack even if the underlying function is not a weak one.

Our above observation can be used as-is, or in conjunction with the method we describe in the following section. We note that the data and time complexities required for the generation of the sequence are discussed separately, as they depend heavily on the structure of the analyzed cipher.[5]

3.2 A New Approach for Attacking the Underlying Round Functions

The method described in the previous section enables the attacker to produce many related-key plaintext pairs given one such pair. As noted earlier, these pairs can be used to mount any regular key recovery attack on the underlying round function. However, in many cases, no such attacks exist, while there is a related-key attack on the underlying round function, e.g., a related-key differential attack.

Our new framework allows to combine the related-key structure with a related-key attack on the underlying function. The main feature of the new framework is examining and comparing several chains of related-key plaintext pairs encrypted under different (but related) pairs of related keys.

Recall that $E_k = f_{k_r} \circ f_{k_{r-1}} \circ \cdots \circ f_{k_1}$, and assume that there exists a related-key attack on $f(\cdot)$. We shall describe the case of a related-key attack that requires two related keys, but related-key attacks which require more keys can be easily integrated into this framework.

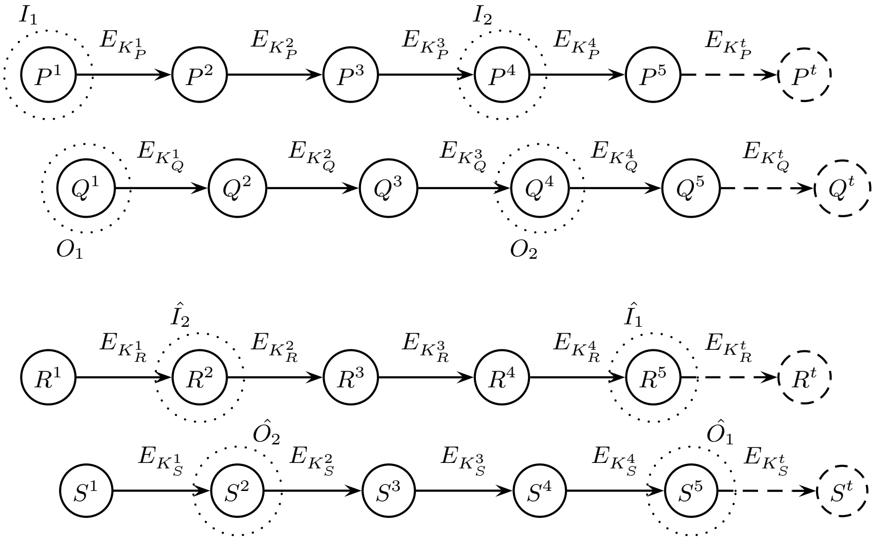
Assume that the related-key attack on $f(\cdot)$ uses two related keys k_1 and $\hat{k_1}$. We denote the data used in the attack by the set of input/output pairs $(I_1, O_1), (I_2, O_2), \ldots$ encrypted under k_1, and the set of input/output pairs $(\hat{I_1}, \hat{O_1}), (\hat{I_2}, \hat{O_2}), \ldots$ encrypted under $\hat{k_1}$. There might be some relation between the inputs, e.g., in the case of a related-key differential attack, the relation between the inputs is $I_j \oplus \hat{I_j} = \Delta_{IN}$.

Our unified attack allows to mount the attack on $f_{k_1}(\cdot)$ although the outputs of $f_{k_1}(\cdot)$ are not immediately available to the attacker. The outputs are detected as the related-key plaintext counterparts of the inputs (if more than one input/output pair is needed under a single pair of keys, they can be generated using the approach provided in Section 3.1).

The basic algorithm of the unified attack is the following:

1. Pick two plaintexts P and R. For every possible pair of plaintexts Q and S, perform the following:
   (a) Assume that (P, Q) is a related-key plaintext pair with respect to the keys (K_P, K_Q) and generate from them a chain of related-key plaintext pairs (P^t, Q^t) with respect to the same pair of keys.

[5] For the sake of simplicity, we describe only attacks that use pairs encrypted under the same pair of related keys. In some cases, the attack can be improved by aggregating the information obtained from several pools of related-key plaintext pairs encrypted under different pairs of related keys.



(P^i, Q^i) are related-key plaintext pairs with respect to (K_P^i, K_Q^i).
(R^j, S^j) are related-key plaintext pairs with respect to (K_R^j, K_S^j).
(I_i, O_i) are input/output pairs for the related-key attack on $f(\cdot)$ for the first key.
$(\hat{I}_i, \hat{O}_i)$ are input/output pairs for the related-key attack on $f(\cdot)$ for the second key.

Fig. 2. Overview of the Related-Key Plaintext Pairs Used in the Unified Attack

(b) Assume that (R, S) is a related-key plaintext pair with respect to the keys (K_R, K_S) and generate from them a chain of related-key plaintext pairs (R^m, S^m) with respect to the same pair of keys.

(c) Detect a set of (P^{t_i}, Q^{t_i}) such that $P^{t_i} = I_i$, and let $O_i = Q^{t_i}$.

(d) Detect a set of (R^{m_j}, S^{m_j}) such that $R^{m_j} = \hat{I}_j$, and let $\hat{O}_j = S^{m_j}$.

(e) Apply the related-key attack on $f_{k_1}(\cdot)$ and $f_{\hat{k_1}}(\cdot)$ using the inputs $I_1, I_2, \ldots$ and $\hat{I}_1, \hat{I}_2, \ldots$ and the corresponding outputs $O_1, O_2, \ldots$ and $\hat{O}_1, \hat{O}_2, \ldots$.

2. If for all the checked pairs Q and S the related-key attack on f_{k_1} fails, repeat Step 1 with a different choice of P and R.

The unified attack considers simultaneously two chains $\{(P^t, Q^t)\}$ and $\{(R^m, S^m)\}$ encrypted under different pairs of related keys. Each chain contains a set of input/output pairs for the round functions f_{k_1} and $f_{\hat{k_1}}$, thus allowing to apply the related-key attack. We outline these chains of plaintexts and their relations in Figure 2.

Our new approach increases the problem of identifying the related-key plaintext pairs. Now, the attacker has to find two related-key plaintext pairs (P, Q) and (R, S) rather than only one.

The general algorithm can be improved in many cases. The relations between I_i's and O_i's can be used to reduce the number of related-key plaintext counterparts corresponding to P and (independently) to R [3,13]. For example, if $f(\cdot)$ is one round of a Feistel cipher, then the number of possible counterparts of P (and of R) is greatly reduced. If there exists a relation between the values of O_i's and the values of $\hat{O}_j$'s they can be used as well to reduce the need of trying all possible pairs of pairs ((P, Q) and (R, S)). We observe that the possible number of counterparts can further be reduced using relations between the chains. For example, assume that there exists a related-key differential of $f(\cdot)$ that predicts that with high probability the input difference α becomes an output difference β. In this case, obtaining the first related-key plaintext pair (P, Q) suggests that with a high probability $(R = P \oplus \alpha, S = Q \oplus \beta)$ is also a related-key plaintext pair. We note that in the case of the slide attack, a similar improvement is suggested in [14].

3.3 Comparison with Other Related-Key Attacks

The main drawback of our proposed framework is the fact that the new attack requires encryption under multiple related keys. Hence, in order to measure the effectiveness of the new framework, it is not sufficient to compare it with the classic generic attacks, such as exhaustive key search and dictionary attacks. The framework should be compared also to generic attacks that allow the attacker to use encryption under multiple related keys. In this section we consider two attacks of this class.

The first attack is a generic time-memory-key trade-off attack suggested in [11]. In the classic time-memory trade-off attack on block ciphers, if the number of keys is N, the available memory is M, and the time complexity of the online step of the attack is T, then $N^2 = TM^2$. In addition, the attack requires a precomputation step of N operations. In [11] the authors show that if the attacker is able to ask for encryptions under D related keys, then the complexities of the attack can be reduced according to the curve $(N/D)^2 = TM^2$. The length of the precomputation step is also reduced to N/D.

In view of this generic attack, it seems that an attack requiring encryption under D related keys should be compared to an exhaustive search over a space of N/D keys or to a classic time-memory tradeoff attack over such key space.[6]

The second attack is the generic attack presented in [2]. The attack uses the fact that if for any block cipher a key is periodic, i.e., can be rotated to itself, this can be identified easily using a few related-key queries. Actually, this property defines a weak key class for all block ciphers. In the attack, the attacker asks for the encryption under various keys, tracing the relation of the keys to the original key, and checks whether the related keys fall into the weak key class.

We note that the attack can be applied with other weak key classes as well. In general, if the size of a weak key class is WK and the total number of possible

[6] We note that the time-memory-key attack recovers only one of the related keys. However, the other keys can be easily found using the relations between the related keys. Also note that if the relation between the keys is correlated to the tables constructed in the time-memory-key attack, the attack might fail.

keys is N, the attack is expected to require N/WK related keys. For the generic weak-key class presented in [2], the attack requires $2^{N/2}$ related keys and a few chosen plaintext queries under each of the keys.

The attack presented in Section 4.2 requires 256 related keys, has a memory complexity of 2^{66} and a time complexity of 2^{100} for a 128-bit key cipher. Hence, its complexity is better than that of the corresponding time-memory-key tradeoff attack. It also compares favorably with the generic attack presented in [2] since for such a small amount of related keys the success probability of the generic attack is 2^{-56}.

4 Attacking 4*r*-Round IDEA

IDEA is a 64-bit, 8.5-round block cipher with 128-bit keys [37]. IDEA is a composition of XOR operations, additions modulo 2^{16}, and multiplications over the field $GF(2^{16}+1)$. The full description of IDEA is given in Appendix A.

4.1 Observations on IDEA Used in the Attack

Our attack on IDEA is based on the following two observations:

1. The key schedule of IDEA has the following property: If the original key is rotated by 75 bits to the left and entered into the key schedule algorithm, the resulting subkeys of rounds 1–4 are the same as the subkeys of rounds 5–8 for the original key. Hence, we can treat $4r$-IDEA as a cascade of r 4-round IDEA components.
2. There exists a related-key truncated differential on 4-round IDEA. The key difference of the differential is in bits 25 and 48. The input difference of the differential is $\Delta_{IN} = (0, 8040_x, 0, 0)$ and the fourth input word is set to 1. This input difference leads to an output difference $\Delta_{OUT} = (a, a, b, b)$, where a and b are some (not necessarily different) 16-bit values, with probability of 2^{-17}. The related-key truncated differential is fully described in Appendix B.

We denote four rounds of IDEA with key k (i.e., the first 128 bits that are used as subkeys), by $4IDEA_k$. Thus, a $4r$-round IDEA with a key K can be described as

$$E_K(P) = 4IDEA_{K \lll 75\cdot(r-1)}(\ldots(4IDEA_{K\lll 75}(4IDEA_K(P)))),$$

where $\lll$ is the rotate left operation. The attack on $4r$-round IDEA uses the unified related-key framework. The cipher is treated as a cascade of r 4-round components, and the related-key truncated differential is used to attack the underlying function, i.e., 4-round IDEA.

A pair of ciphertexts $C^1 = (C^1_1, C^1_2, C^1_3, C^1_4)$ and $C^2 = (C^2_1, C^2_2, C^2_3, C^2_4)$ that satisfy the output difference Δ_{OUT} satisfy that

$$C^1_1 \oplus C^2_1 = C^1_2 \oplus C^2_2 \quad \text{and} \quad C^1_3 \oplus C^2_3 = C^1_4 \oplus C^2_4$$

These relations can be easily rewritten into:

$$C_1^1 \oplus C_2^1 = C_1^2 \oplus C_2^2 \quad \text{and} \quad C_3^1 \oplus C_4^1 = C_3^2 \oplus C_4^2$$

Thus, we define the function *evaluate*(C), to efficiently help us to determine right pairs:

$$evalute(C = (C_1, C_2, C_3, C_4)) = C_1 \oplus C_2 || C_3 \oplus C_4.$$

Thus, in order to check whether two values (C^1, C^2) satisfy the output difference of the related-key differential, it is sufficient to check whether $evaluate(C^1) = evaluate(C^2)$.

For the sake of simplicity we shall describe the attack on 8-round IDEA. The changes needed for attacking $4r$-round IDEA with $r \neq 2$ are relatively small (and are mainly in the data generation phase).

The attack has three main steps. The first one is data generation, where chains of plaintexts are generated according to the framework we described in the previous section. The purpose of the chains is to produce several related-key plaintext pairs simultaneously. The second step is composed of analyzing the chains and trying to find the related-key plaintext pairs efficiently. This step is performed by using the key recovery step (the third one). The last step is the key recovery step, in which the candidates for being related-key plaintext pairs are used for key recovery. Once sufficiently many related-key plaintext pairs are found, so does the right key.

For sake of simplicity we assume that the chains of plaintexts generated in the attack compose the entire code book, i.e., all plaintexts are there. The cases when this assumption does not hold are discussed in Section 4.4. As long as the chains contain enough related-key plaintext pairs, our attack works. To justify the assumption we made, we note that starting with a chain which does not contain enough plaintexts is highly unlikely.

4.2 The Attack Algorithm

1. **Data Generation**
   (a) Let K_P be the unknown key, and let $K_Q = K_P \lll 75$. Let $K_R = K_P \oplus e_{25,48}$, i.e., K_R is the same as K_P in all bits but bits 25 and 48. Finally, let $K_S = K_R \lll 75$. Pick randomly four plaintexts: $P_0^0, Q_0^{75}, R_0^0, S_0^{75}$.
   (b) Starting from P_0^0 and K_P compute the following chain:

$$P_i^{l+22 \bmod 128} = \begin{cases} E_{K_P \lll l}(P_i^\ell) & \text{if } l + 22 \not\equiv 0 \bmod 128 \\ E_{K_P \lll l}(P_{i-1}^\ell) & \text{if } l + 22 \equiv 0 \bmod 128 \end{cases}$$

   Continue till P_0^0 is about to be encrypted under K_P again (according to our assumption — after 2^{64} encryptions under K_P). We denote this chain by $Chain_P$.
   (c) Compute $Chain_Q$ starting from Q_0^{75} as the plaintext and K_Q as the key, using the same process.

(d) Denote by $K_R = K_P \oplus e_{25,48}$, Pick a plaintext R_0^0 randomly, and repeat the previous step with the key K_R to obtain the chain $Chain_R = R_0^0, R_0^{22}, \ldots, R_{2^{64}-1}^{106}$.

(e) Pick a plaintext S_0^{75} randomly and perform the same operation in the previous step with the key $K_S = K_R \lll 75$, obtaining the chain $Chain_S = S_0^{75}, S_0^{97}, \ldots, S_1^{75}, \ldots, S_{2^{64}-1}^{53}$. For sake of simplicity, we assume that each of the four chains covers all possible plaintexts (i.e., each plaintext is encrypted under every key of the chain). We deal with the case of several chains in Section 4.4.

2. **Analyzing the Chains:** Locate a set of 2^{36} pairs of plaintexts $(P_{i_1}^0, P_{i_2}^0)$ in $Chain_P$ whose fourth word equals 1 for both plaintexts. For each such pair:
   (a) Compute the values of j_1 and j_2, such that $R_{j_1}^0 = P_{i_1}^0 \oplus \Delta_{IN}$ and $R_{j_2}^0 = P_{i_2}^0 \oplus \Delta_{IN}$.
   (b) For each $S_m^{75} \in Chain_S$ store the 64-bit value $value_S = evaluate(S_m^{75}) || evaluate(S_{m+j_2-j_1}^{75})$ along with m in a table $Table_S$ indexed by the computed value.
   (c) For each $Q_l^{75} \in Chain_Q$ perform:
      – Compute the 64-bit value $value_Q = evaluate(Q_l^{75}) || evaluate (Q_{l+i_2-i_1}^{75})$. Search for $value_Q$ in $Table_S$.
      – For each possible value of m associated with $value_Q$, check whether

$$Q_l^{75} = 4IDEA_{K_P}(P_{i_1}^0); \qquad Q_{l+i_2-i_1}^{75} = 4IDEA_{K_P}(P_{i_2}^0);$$

$$S_m^{75} = 4IDEA_{K_R}(R_{j_1}^0); \qquad S_{m+j_2-j_1}^{75} = 4IDEA_{K_R}(R_{j_2}^0)$$

using the key recovery attack that uses the respective pairs (we outline this attack later), where 4IDEA denotes 4-round IDEA. If the key recovery attack succeeds, the key is found.

$Chain_P$ and $Chain_Q$ contains input/output pairs to $4IDEA_{K_P}(\cdot)$ whose order is unknown. The same is true for $Chain_R$ and $Chain_S$ with respect to $4IDEA_{K_R}(\cdot)$. The attack tries all the possible shifts between $Chain_P$ and $Chain_Q$ (for which one is the correct shift). To prevent the need of checking all the shifts between $Chain_R$ and $Chain_S$, we use the related-key truncated differential. The key difference between K_P and K_R is the key difference of the differential, which means that an input pair $(P_i^0, R_j^0 = P_i^0 \oplus \Delta_{IN})$ is more likely to have the corresponding outputs (Q_l^{75}, S_m^{75}) satisfying the output difference of the differential.

The attack algorithm first tries to find the shift between $Chain_P$ and $Chain_Q$. For each such shift, we assume it is the correct one and we find 2^{36} pairs of pairs $(P_{i_1}^0, R_{j_1}^0)$ and $(P_{i_2}^0, R_{j_2}^0)$ with difference Δ_{IN}. If indeed the shift was correct, the probability that the corresponding outputs satisfy the output difference (twice) is 2^{-34}, and thus, the only remaining problem is finding the corresponding outputs. For the $Chain_P$, as we know the shift of $Chain_Q$ we know the outputs. Thus, we only need to find the shift of $Chain_S$ with respect to $Chain_R$.

The last task is achieved by observing that if S_m^{75} is the output of R_{j_1}, then $S_{m+j_2-j_1}^{75}$ is the output of R_{j_2}. Thus, we compute for each S_m^{75} the value of

$evaluate(S^{75}_m)||evalaute(S^{75}_{m+j_2-j_1})$. Similarly, if Q^{75}_l is the output of $P^0_{i_1}$, then $Q^{75}_{l+i_2-i_1}$ is the output of $P^0_{i_2}$. If indeed the pairs $(P^0_{i_1}, R^0_{j_1})$ and $(P^0_{i_2}, R^0_{j_2})$ satisfy the differential, then it must hold that $evaluate(S^{75}_m) = evalute(Q^{75}_l)$ and that $evalaute(S^{75}_{m+j_2-j_1}) = evalaute(Q^{75}_{l+i_2-i_1})$. Thus, the attack succeeds in retrieving the right shift of $Chain_S$ with respect to $Chain_R$ given the right shift of $Chain_Q$ with respect to $Chain_P$.

The most basic attack retrieves the subkey bits involved in the last MA layer (by finding the key value for which both pairs have a zero difference before the MA layer). An additional fast filtering of wrong subkey values can be performed using the first round of the truncated differential (verifying that indeed during the first KA layer the difference in the second word becomes 8000_x).

If indeed the related-key plaintext pairs are the ones analyzed, then there is a probability of 2^{-34} that the attack succeeds (as both pairs should be right pairs for the attack to succeed). If this is not the case, then it is highly unlikely that the key recovery attack succeeds. We first note that for any value of Q^{75}_l we expect about one suggestion for the value of m. Then, the probability that two wrong pairs (or even one wrong pair and one right pair) agree on the key for the fourth round MA layer is 2^{-32}. Considering the additional filtering based on the first round as well, the probability that a wrong Q^{75}_l leads to a consistent key suggestion is 2^{-34}.

Thus, it is expected that of the 2^{64} possible relations for a given $P^0_{i_1}$ and $P^0_{i_2}$, only 2^{30} values of Q^{75}_l may seem suitable, and these can be easily discarded using an additional "pair" of related-key plaintexts and ciphertexts.

4.3 The Time Complexity of the Attack

The first step of the attack is composed of constructing four chains. Each such chain requires the encryption of 2^{64} values, each under 64 keys. Thus, the time complexity of the data generation is $4 \cdot 64 \cdot 2^{64} = 2^{72}$ encryptions.

The time complexity of Step 2 is mostly dominated by Step 2(c). It is easy to see that using a well chosen data structure, Step 2(a) takes a relatively small number of operations for each pair. For each of the 2^{36} pairs $(P^0_{i_1}, P^0_{i_2})$, the time complexity of Step 2(b) is about 2^{64} operations. Step 2(c) is repeating 2^{64} times a basic operation and a key recovery attack. The key recovery attack can be efficiently simulated using two table lookups (to a table suggesting for each pair the respective MA layer subkey). Thus, the time complexity of Step 2 is about $2^{36+64} = 2^{100}$ operations.[7]

We conclude that the attack requires 2^{72} related-key chosen plaintexts, and has a time complexity of about 2^{100} operations.

The memory complexity of the attack is dominated by the stored data and the table containing $value_S$. Thus, the total memory required by our attack is at

[7] By generating the indices using a little different order, it is possible to use eight 64-bit logical operations for the computation of $value_Q$ or $value_S$. Thus, the term "operation" refers here to about eighteen 64-bit logical operations and seven memory accesses.

most $4 \cdot 2^{64}$ blocks of memory for the data, and additional 2^{64} entries of the form $m||value_S$ (which take two blocks of memory each). Thus, the total memory used by the attack is at most $48 \cdot 2^{64}$ bytes (which are $6 \cdot 2^{64}$ blocks of 64 bits).

4.4 Dealing with Multiple Chains

When the chains do not cover the entire plaintext space, then the attack algorithm has to be tweaked a bit to ensure success. We first note that we can divide the entire plaintext space into multiple chains, i.e., $Chain_{P_1}$, $Chain_{P_2}$, ..., and equivalent $Chain_{Q_1}$, $Chain_{Q_2}$, ... Then, we wait till 2^{36} pairs of values $(P^0_{i_1}, P^0_{i_2})$ that can be used in the attack are encountered in some chain $Chain_{P_n}$.

We can treat the chains as generated by a random permutation over the plaintext space of a given key, e.g., K_P. Hence, the analysis of [25] can be applied, revealing that the longest chain is expected to cover about 2^{63} plaintexts (for a given key), that the second longest chain covers about 2^{62} plaintexts, etc.

As in the original description, we start from a random plaintext and start generating the chain. With high probability this plaintext belongs to one of the longest chains. Doing the same for the generation of the other chains, we are expected to find chains of the same length, i.e., $Chain_{P_i}$ and $Chain_{Q_i}$ of the same length, and $Chain_{R_j}$ and $Chain_{S_j}$ of the same length. As the existence of a related key plaintext pair in the two chains $Chain_{P_i}$ and $Chain_{Q_i}$ can happen only if their length is the same, then this can improve the attack, by first generating the chains, and then reducing the number of candidate related-key plaintext pairs by taking the chain lengths into account.

There is a small problem that may rise. In order for the attack to work, $(P^0_{i_1}, P^0_{i_2})$ counterparts, i.e., $R^0_{j_1}$ and $R^0_{j_2}$ has to be in the same chain $chainR_j$ (for the indexing phase done in Step 2(b)). This condition cannot be assured. However, assuming that the cycle structure of the chains $Chain_{R_1}$, $Chain_{R_2}$, ... behaves as if 8-round IDEA is a random permutation, it is expected that there is a chain $Chain_R$ whose size is larger than 2^{63} with overwhelming probability [25]. If this is the case, we can change the attack algorithm such that only pairs of plaintexts $(P^0_{i_1}, P^0_{i_2})$ whose counterparts are in that chain, are considered. This increases the number of pairs that are considered from 2^{36} to 2^{38} at most, but as at least 3/4 of the new pairs are not analyzed, this does not increase the time complexity of the attack.

Actually, the time complexity of the attack is expected to drop by a factor of about 2. This is caused by the fact that the indexing is now performed in some $Chain_S$ (the one corresponding to $Chain_R$) whose size is smaller than 2^{64}. Also, there are less candidates for Q^{75}_l that need to be considered.

We note that we can use even shorter chains, as long as there are sufficiently many candidate pairs between the chains $Chain_{P_i}$ and $Chain_{R_j}$. hence, the attacker starts to generate $Chain_{P_i}$ until he obtains the three longest chains. Then, the attacker has to find a $Chain_Q$ with any of these lengths (which can be easily done, as the probability of picking a plaintext at random from the longer chains is significantly higher). The same is done for $Chain_{R_j}$ and $Chain_S$, where we stop once we have sufficiently long chains (and thus, enough candidate pairs for the analysis).

Thus, in this case our attack requires about 2^{99} 8-round IDEA encryptions. The data complexity can be slightly reduced as well. For generating $Chain_{P_i}$ and $Chain_{R_j}$ we do not need to cover all the small chains. Hence, it is expected that generating these chains requires roughly $2^{64}-2^{52}$ adaptive chosen plaintexts and ciphertexts encrypted under 64 keys each. For $Chain_{Q_i}$ and $Chain_{S_j}$ we need only to find one of the longer chains, and thus it is expected that $2^{63.4}$ adaptively chosen plaintexts and ciphertexts (encrypted under 64 keys each) are required. Thus, the data complexity of the attack is $2 \cdot 2^{70} + 2 \cdot 2^{69.4} = 2^{71.7}$ related-key adaptive chosen plaintexts and ciphertexts.

We note that this attack requires more data than the entire codebook for a given key. However, for any given key of the 256 involved keys, we do not require the entire code book.

4.5 Changes for $4r$-Round IDEA with $r \neq 2$

As mentioned earlier, it is possible to apply our attack to $4r$-IDEA when the number of rounds is much larger than 8. The changes in the attack algorithm are only in the data generation phase. While for 8-round IDEA, the chains are constructed as $Chain_P = P_0^0, P_0^{22}, \ldots$, for $4r$-round IDEA the chains are of the form $Chain_P = P_0^0, P_0^{75 \cdot r \bmod 128}, P_0^{75 \cdot r \cdot 2 \bmod 128}, P_0^{75 \cdot r \cdot 3 \bmod 128}, \ldots$.

When $gcd(r, 128) = 2$, the obtained attack has the same data complexity as well as the same number of involved keys (64 for each of the chains). Besides the data generation phase, the attack is the same.

For the cases when $gcd(r, 128) = 1$ the chains $Chain_P$ and $Chain_Q$ are actually the same chain (and $Chain_R$ and $Chain_S$ as well). This follows the fact that for such values of r, K_P is rotated each time to the left by a number of bits which is eventually equal to 75 (i.e., there exists g s.t. $75 \cdot r \cdot g \equiv 75 \bmod 128$). Again, this has no affect on any other steps of the attack or on its date complexity.

When $gcd(r, 128) > 2$, the all the chains are shorter, as the number of keys needed for closing the cycle of the key schedule algorithm is shorter than 64. In that case, the data complexity of the attack drops by a factor of $gcd(r, 128)/2$, as well as the number of keys. For example, for $r = 128$, the data complexity of the attack is only 2^{66} plaintexts, and the number of keys is reduced to four.

5 Summary and Conclusions

Our new framework combines the various kinds of related-key attacks. We show that by combining them, we create a powerful attack. For example, we present the first attack on $4r$-round IDEA, and in particular, the first published work that can break 8 rounds of IDEA. The complexities of our new attack on IDEA, along with the best previously known attacks, are summarized in Table 1.

Table 1. Selected Known Attacks on IDEA and Our New Results

Rounds	Attack Type	Complexity			# of Keys	Source
		Data	Time	Memory		
5	Differential-Linear	16 KP	2^{114}	32	1	[9]
5.5	Higher-Order Diff.-Lin.	2^{34} CP	$2^{126.8}$	2^{35}	1	[9]
6	Higher-Order Diff.-Lin.	$2^{64} - 2^{52}$ KP	$2^{126.8}$	2^{64}	1	[9]
7	Related-Key Rectangle	2^{65} RK-CP	$2^{104.2}$	2^{66}	4	[8]
7.5	Related-Key Linear	$2^{43.5}$ RK-KP	$2^{115.1}$	$2^{44.5}$	2	[8]
8	Unified Related-Key	2^{72} RK-KP	2^{100}	$2^{66.6}$	256	Section 4
8	Unified Related-Key	$2^{71.7}$ RK-ACP	2^{99}	$2^{66.6}$	256	Section 4.4
8	Unified Related-Key	2^{71} RK-ACP	2^{103} MA	$2^{66.6}$	256	Appendix C
$4r$	Unified Related-Key	$2^{72}/x$ RK-KP	2^{100}	$2^{66.6}$	$256/x$	Section 4

$x = \lceil (gcd(r, 128)/2 \rceil$

KP – Known plaintext, CP – Chosen plaintext, RK – Related key,
ACP – Adaptive chosen plaintexts, MA – Memory accesses.
Time complexity is measured in encryption units (unless stated otherwise).
Memory complexity is measured in blocks.

References

1. Ayaz, E.S., Selçuk, A.A.: Improved DST Cryptanalysis of IDEA. In: Biham, E., Youssef, A.M. (eds.) SAC 2006. LNCS, vol. 4356, pp. 1–14. Springer, Heidelberg (2007)
2. Bellare, M., Kohno, T.: A Theoretical Treatment of Related-Key Attacks: RKA-PRPs, RKA-PRFs, and Applications. In: Biham, E. (ed.) EUROCRYPT 2003. LNCS, vol. 2656, pp. 491–506. Springer, Heidelberg (2003)
3. Biham, E.: New Types of Cryptanalytic Attacks Using Related Keys. Journal of Cryptology 7(4), 229–246 (1994)
4. Biham, E., Biryukov, A., Shamir, A.: Miss in the Middle Attacks on IDEA and Khufu. In: Knudsen, L.R. (ed.) FSE 1999. LNCS, vol. 1636, pp. 124–138. Springer, Heidelberg (1999)
5. Biham, E., Dunkelman, O., Keller, N.: The Rectangle Attack – Rectangling the Serpent. In: Pfitzmann, B. (ed.) EUROCRYPT 2001. LNCS, vol. 2045, pp. 340–357. Springer, Heidelberg (2001)
6. Biham, E., Dunkelman, O., Keller, N.: New Combined Attacks on Block Ciphers. In: Gilbert, H., Handschuh, H. (eds.) FSE 2005. LNCS, vol. 3557, pp. 126–144. Springer, Heidelberg (2005)
7. Biham, E., Dunkelman, O., Keller, N.: Related-Key Boomerang and Rectangle Attacks. In: Cramer, R.J.F. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 507–525. Springer, Heidelberg (2005)
8. Biham, E., Dunkelman, O., Keller, N.: New Cryptanalytic Results on IDEA. In: Lai, X., Chen, K. (eds.) ASIACRYPT 2006. LNCS, vol. 4284, pp. 412–427. Springer, Heidelberg (2006)
9. Biham, E., Dunkelman, O., Keller, N.: A New Attack on 6-Round IDEA. In: Biryukov, A. (ed.) FSE 2007. LNCS, vol. 4593, pp. 211–224. Springer, Heidelberg (2007)

10. Biham, E., Shamir, A.: Differential Cryptanalysis of the Data Encryption Standard. Springer, Heidelberg (1993)
11. Biryukov, A., Mukhopadhyay, S., Sarkar, P.: Improved Time-Memory Trade-Offs with Multiple Data. In: Preneel, B., Tavares, S. (eds.) SAC 2005. LNCS, vol. 3897, pp. 110–127. Springer, Heidelberg (2006)
12. Biryukov, A., Nakahara Jr., J., Preneel, B., Vandewalle, J.: New Weak-Key Classes of IDEA. In: Deng, R.H., Qing, S., Bao, F., Zhou, J. (eds.) ICICS 2002. LNCS, vol. 2513, pp. 315–326. Springer, Heidelberg (2002)
13. Biryukov, A., Wagner, D.: Slide Attacks. In: Knudsen, L.R. (ed.) FSE 1999. LNCS, vol. 1636, pp. 245–259. Springer, Heidelberg (1999)
14. Biryukov, A., Wagner, D.: Advanced Slide Attacks. In: Preneel, B. (ed.) EUROCRYPT 2000. LNCS, vol. 1807, pp. 586–606. Springer, Heidelberg (2000)
15. Borisov, N., Chew, M., Johnson, R., Wagner, D.: Multiplicative Differentials. In: Daemen, J., Rijmen, V. (eds.) FSE 2002. LNCS, vol. 2365, pp. 17–33. Springer, Heidelberg (2002)
16. Borst, J., Knudsen, L.R., Rijmen, V.: Two Attacks on Reduced Round IDEA. In: Fumy, W. (ed.) EUROCRYPT 1997. LNCS, vol. 1233, pp. 1–13. Springer, Heidelberg (1997)
17. Brown, L., Seberry, J.: Key Scheduling in DES Type Cryptosystems. In: Seberry, J., Pieprzyk, J.P. (eds.) AUSCRYPT 1990. LNCS, vol. 453, pp. 221–228. Springer, Heidelberg (1990)
18. Daemen, J., Govaerts, R., Vandewalle, J.: Cryptanalysis of 2.5 Rounds of IDEA (Extended Abstract), technical report 93/1, Department of Electrical Engineering, ESAT–COSIC, Belgium (1993)
19. Daemen, J., Govaerts, R., Vandewalle, J.: Weak Keys for IDEA. In: Stinson, D.R. (ed.) CRYPTO 1993. LNCS, vol. 773, pp. 224–231. Springer, Heidelberg (1994)
20. Daemen, J., Knudsen, L.R., Rijmen, V.: The Block Cipher Square. In: Biham, E. (ed.) FSE 1997. LNCS, vol. 1267, pp. 149–165. Springer, Heidelberg (1997)
21. Demirci, H.: Square-like Attacks on Reduced Rounds of IDEA. In: Nyberg, K., Heys, H.M. (eds.) SAC 2002. LNCS, vol. 2595, pp. 147–159. Springer, Heidelberg (2003)
22. Demirci, H., Selçuk, A.A., Türe, E.: A New Meet-in-the-Middle Attack on the IDEA Block Cipher. In: Matsui, M., Zuccherato, R.J. (eds.) SAC 2003. LNCS, vol. 3006, pp. 117–129. Springer, Heidelberg (2004)
23. Ferguson, N., Kelsey, J., Lucks, S., Schneier, B., Stay, M., Wagner, D., Whiting, D.: Improved Cryptanalysis of Rijndael. In: Schneier, B. (ed.) FSE 2000. LNCS, vol. 1978, pp. 213–230. Springer, Heidelberg (2001)
24. Furuya, S.: Slide Attacks with a Known-Plaintext Cryptanalysis. In: Kim, K.-c. (ed.) ICISC 2001. LNCS, vol. 2288, pp. 214–225. Springer, Heidelberg (2002)
25. Granville, A.: Cycle lengths in a permutation are typically Poisson distributed, http://www.dms.umontreal.ca/~andrew/PDF/CycleLengths.pdf
26. Hawkes, P.: Differential-Linear Weak Keys Classes of IDEA. In: Nyberg, K. (ed.) EUROCRYPT 1998. LNCS, vol. 1403, pp. 112–126. Springer, Heidelberg (1998)
27. Hawkes, P., O'Connor, L.: On Applying Linear Cryptanalysis to IDEA. In: Kim, K.-c., Matsumoto, T. (eds.) ASIACRYPT 1996. LNCS, vol. 1163, pp. 105–115. Springer, Heidelberg (1996)
28. Hong, S., Kim, J., Kim, G., Lee, S., Preneel, B.: Related-Key Rectangle Attacks on Reduced Versions of SHACAL-1 and AES-192. In: Gilbert, H., Handschuh, H. (eds.) FSE 2005. LNCS, vol. 3557, pp. 368–383. Springer, Heidelberg (2005)

29. Iwata, T., Kohno, T.: New Security Proofs for the 3GPP Confidentiality and Integrity Algorithms. In: Roy, B., Meier, W. (eds.) FSE 2004. LNCS, vol. 3017, pp. 427–445. Springer, Heidelberg (2004)
30. Jakimoski, G., Desmedt, Y.: Related-Key Differential Cryptanalysis of 192-bit Key AES Variants. In: Matsui, M., Zuccherato, R.J. (eds.) SAC 2003. LNCS, vol. 3006, pp. 208–221. Springer, Heidelberg (2004)
31. Junod, P.: New Attacks Against Reduced-Round Versions of IDEA. In: Gilbert, H., Handschuh, H. (eds.) FSE 2005. LNCS, vol. 3557, pp. 384–397. Springer, Heidelberg (2005)
32. Kelsey, J., Schneier, B., Wagner, D.: Key-Schedule Cryptoanalysis of IDEA, G-DES, GOST, SAFER, and Triple-DES. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 237–251. Springer, Heidelberg (1996)
33. Kelsey, J., Schneier, B., Wagner, D.: Related-Key Cryptanalysis of 3-WAY, Biham-DES, CAST, DES-X, NewDES, RC2, and TEA. In: Han, Y., Quing, S. (eds.) ICICS 1997. LNCS, vol. 1334, pp. 233–246. Springer, Heidelberg (1997)
34. Kim, J., Hong, S., Preneel, B.: Related-Key Rectangle Attacks on Reduced AES-192 and AES-256. In: Biryukov, A. (ed.) FSE 2007. LNCS, vol. 4593, pp. 225–241. Springer, Heidelberg (2007)
35. Kim, J., Kim, G., Hong, S., Hong, D.: The Related-Key Rectangle Attack — Application to SHACAL-1. In: Wang, H., Pieprzyk, J., Varadharajan, V. (eds.) ACISP 2004. LNCS, vol. 3108, pp. 123–136. Springer, Heidelberg (2004)
36. Knudsen, L.R.: Cryptanalysis of LOKI91. In: Zheng, Y., Seberry, J. (eds.) AUSCRYPT 1992. LNCS, vol. 718, pp. 196–208. Springer, Heidelberg (1993)
37. Lai, X., Massey, J.L., Murphy, S.: Markov Ciphers and Differential Cryptanalysis. In: Davies, D.W. (ed.) EUROCRYPT 1991. LNCS, vol. 547, pp. 17–38. Springer, Heidelberg (1991)
38. Matsui, M.: Linear Cryptanalysis Method for DES Cipher. In: Helleseth, T. (ed.) EUROCRYPT 1993. LNCS, vol. 765, pp. 386–397. Springer, Heidelberg (1994)
39. Meier, W.: On the Security of the IDEA Block Cipher. In: Helleseth, T. (ed.) EUROCRYPT 1993. LNCS, vol. 765, pp. 371–385. Springer, Heidelberg (1994)
40. Nakahara Jr., J., Barreto, P.S.L.M., Preneel, B., Vandewalle, J., Kim, H.Y.: SQUARE Attacks Against Reduced-Round PES and IDEA Block Ciphers, IACR Cryptology ePrint Archive, Report 2001/068 (2001)
41. Nakahara Jr., J., Preneel, B., Vandewalle, J.: The Biryukov-Demirci Attack on Reduced-Round Versions of IDEA and MESH Ciphers. In: Wang, H., Pieprzyk, J., Varadharajan, V. (eds.) ACISP 2004. LNCS, vol. 3108, pp. 98–109. Springer, Heidelberg (2004)
42. Raddum, H.: Cryptanalysis of IDEA-X/2. In: Johansson, T. (ed.) FSE 2003. LNCS, vol. 2887, pp. 1–8. Springer, Heidelberg (2003)
43. Tsudik, G., Van Herreweghen, E.: On simple and secure key distribution. In: Conference on Computer and Communications Security, Proceedings of the 1st ACM conference on Computer and communications security, pp. 49–57. ACM Press, New York (1993)
44. Wagner, D.: The Boomerang Attack. In: Knudsen, L.R. (ed.) FSE 1999. LNCS, vol. 1636, pp. 156–170. Springer, Heidelberg (1999)
45. Wheeler, D.J., Needham, R.M.: TEA, a Tiny Encryption Algorithm. In: Preneel, B. (ed.) FSE 1994. LNCS, vol. 1008, pp. 363–366. Springer, Heidelberg (1995)
46. ZDNet, New Xbox security cracked by Linux fans (2002), http://news.zdnet.co.uk/software/developer/0,39020387,2123851,00.htm

A A Description of IDEA

IDEA is a 64-bit, 8.5-round block cipher with 128-bit keys proposed by Lai and Massey in 1991 [37].

Each round of IDEA (besides the last one) consists of two layers. Let the input of round i be denoted by four 16-bit words $(X_1^i, X_2^i, X_3^i, X_4^i)$. The first layer, denoted by KA, affects each word independently: The first and the fourth words are multiplied by subkey words (mod $2^{16}+1$) where 0 is replaced by 2^{16}, and the second and the third words are added with subkey words (mod 2^{16}). We denote the intermediate values after this half-round by $(Y_1^i, Y_2^i, Y_3^i, Y_4^i)$. Let Z_1^i, Z_2^i, Z_3^i, and Z_4^i be the four subkey words, then

$$Y_1^i = Z_1^i \odot X_1^i; \quad Y_2^i = Z_2^i \boxplus X_2^i; \quad Y_3^i = Z_3^i \boxplus X_3^i; \quad Y_4^i = Z_4^i \odot X_4^i,$$

where $\odot$ denotes multiplication modulo $2^{16}+1$ with 0 replaced by 2^{16}, and where $\boxplus$ denotes addition modulo 2^{16}.

The second layer, denoted by MA, accepts two 16-bit words p^i and q^i computed as $(p^i, q^i) = (Y_1^i \oplus Y_3^i, Y_2^i \oplus Y_4^i)$. We denote the two output words of the MA transformation by (u^i, t^i). Denoting the subkey words that enter the MA function of round i by Z_5^i and Z_6^i,

$$t^i = (q^i \boxplus (p^i \odot Z_5^i)) \odot Z_6^i; \quad u^i = (p^i \odot Z_5^i) \boxplus t^i$$

The output of the i-th round is $(Y_1^i \oplus t^i, Y_3^i \oplus t^i, Y_2^i \oplus u^i, Y_4^i \oplus u^i)$. In the last round (round 9) the MA layer is removed. Thus, the ciphertext is $(Y_1^9 || Y_2^9 || Y_3^9 || Y_4^9)$. The structure of a single round of IDEA is shown in Figure 3.

IDEA's key schedule expands the 128-bit key into $6 \cdot 8 + 4 = 52$ subkeys of 16 bits each using a very simple algorithm. The key is first used as the first eight subkeys. Then, the key is rotated by 25 bits to the left, and the outcome is used as the next eight subkeys. The rotation by 25 bits to the left is repeatedly used as many times as needed. All the subkeys for 8.5-round IDEA are listed in Table 2.

B A 4-Round Related-Key Truncated Differential of IDEA

Our attack exploits a 4-round related-key truncated differential. This related-key differential has probability 2^{-17} when a 16-bit condition on the plaintext is imposed, as we describe later.

The differential holds for rounds 1–4 of IDEA, and thus can be used as a building block in the unified related-key attack. The key difference is in bits 25 and 48 (i.e., the two related keys K^1 and K^2 satisfy $K^1 \oplus K^2 = \Delta K = e_{25,48}$).

The input difference of the differential is $\Delta_{IN} = (0, 8040_x, 0, 0)$, and the fourth input words of both plaintexts are required to be 1. Thus, after the first key addition layer, the difference becomes $(0, 8000_x, 0, 8000_x)$ with probability 1/2. We note that the difference in the most significant bit of the fourth word is

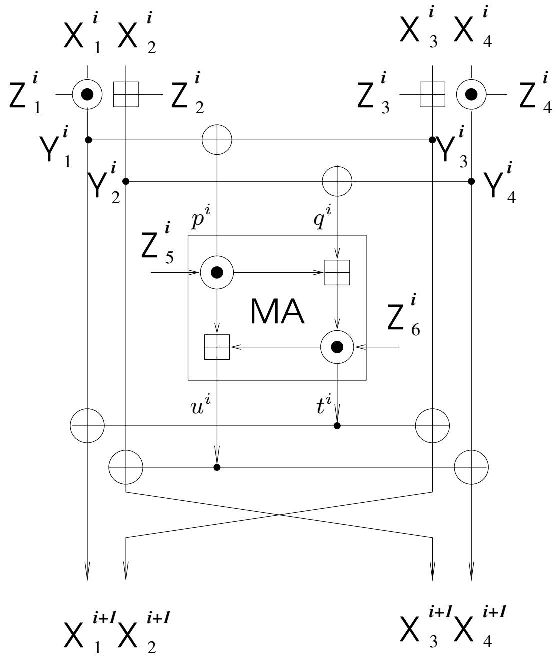

X_1^i X_2^i X_3^i X_4^i
Z_1^i Z_2^i Z_3^i Z_4^i
Y_1^i Y_2^i Y_3^i Y_4^i
p^i q^i
Z_5^i
MA
Z_6^i
u^i t^i
X_1^{i+1} X_2^{i+1} X_3^{i+1} X_4^{i+1}

Fig. 3. One Round of IDEA

Table 2. The Key Schedule Algorithm of IDEA

Round	Z_1^i	Z_2^i	Z_3^i	Z_4^i	Z_5^i	Z_6^i
$i = 1$	0–15	16–31	32–47	48–63	64–79	80–95
$i = 2$	96–111	112–127	25–40	41–56	57–72	73–88
$i = 3$	89–104	105–120	121–8	9–24	50–65	66–81
$i = 4$	82–97	98–113	114–1	2–17	18–33	34–49
$i = 5$	75–90	91–106	107–122	123–10	11–26	27–42
$i = 6$	43–58	59–74	100–115	116–3	4–19	20–35
$i = 7$	36–51	52–67	68–83	84–99	125–12	13–28
$i = 8$	29–44	45–60	61–76	77–92	93–108	109–124
$i = 9$	22–37	38–53	54–69	70–85		

caused by the key difference and the fact that the fourth words of the plaintexts have the value of 1.

The input difference to the first MA layer is $(0, 0)$, and thus, the output difference of the first round is $(0, 0, 8000_x, 8000_x)$. In the second KA layer, the difference in the third word is cancelled by the key difference with probability 1. Under the assumption that the multiplication operation (under the two related subkeys) has a close to random behavior, the difference in the fourth word is cancelled as well with probability 2^{-16}. Hence, with probability 2^{-17} there is a zero difference after the second KA layer.

The zero difference state remains until the MA layer of the fourth round, where the key difference affects the two subkeys. Thus, the output difference of the related-key differential is $\Delta_{OUT} = (a, a, b, b)$ for some unknown a and b.

Note that if the differential is satisfied, then the difference before the MA layer of the fourth round is a zero difference. This property is used in the key deduction phase of the attack presented in Section 4.

C Another Key Recovery Attack on 4r-Round IDEA

In this section we propose a different attack algorithm on 4r-round IDEA. The attack is different than the one in Section 4.2 in the way the chains are used to detect the related-key plaintext pairs. As before, we present the attack on 8-round IDEA, but it can be easily transformed to other 4r-round variants of IDEA.

C.1 Attack Algorithm

1. **Data Generation**
   (a) Pick a plaintext P_0^0 randomly, and ask for its encryption under the unknown key K and for the encryption of the plaintext $R_0^0 = P_0^0 \oplus \Delta_{IN}$ under $K' = K \oplus \Delta K$. Denote the corresponding ciphertexts by P_0^{22} and R_0^{22}, respectively. Then ask for the encryption of P_0^{22} under $K \lll 22$ and of R_0^{22} under $K' \lll 22$, and denote the corresponding ciphertexts by P_0^{44} and R_0^{44}, respectively. Continue the process until the keys are again K and K', and denote the plaintexts by P_1^0 and R_1^0, respectively. Repeat the process until 2^{20} pairs (P_i^0, R_j^0) such that $P_i^0 \oplus R_j^0 = \Delta_{IN}$ with the fourth word of P_i^0 and R_j^0 equal to 1 are encountered. Store each such set of indices (i, j) in a table $Table_{P,R}$. In case the chains end before enough such pairs are encountered, another P_0^0 and R_0^0 are chosen.
   (b) Repeat Steps 1.2 and 1.4 of the attack from Section 4.2 to obtain $Chain_Q$ and $Chain_S$.
2. **Generating Related-Key Plaintext Pairs:** Choose a pool of 2^{32} candidate related-key plaintext counterparts to R_0^0 denoted by $\{S_m^{75}\}_{m=0}^{2^{32}-1}$, such that $\forall m, m'$ the difference $S_m^{75} \oplus S_{m'}^{75}$ is in Δ_{OUT}. Note that the entire space of plaintexts is divided to 2^{32} disjoint pools of this form. For each pool perform the following:
   (a) For each $(i, j) \in Table_{P,R}$, and for all m, compute the encryption of S_m^{75} j positions further in the chain that starts from it, denoted by $S_{m,j}^{75}$. Store in a table the values of $S_{m,j}^{75}$, along with S_m^{75}.
   (b) Compute the set of 2^{32} possible Q_l^{75} such that $S_m^{75} \oplus Q_l^{75} \in \Delta_{OUT}$. (We note that the pool $\{Q_l^{75}\}_{l=0}^{2^{32}-1}$ is actually equal to the pool $\{S_m^{75}\}_{m=0}^{2^{32}-1}$, since Δ_{OUT} is closed under the XOR operation).
   (c) For every $(i, j) \in Table_{P,R}$ and for all l, compute the i'th places in the chain of the encryption of Q_l^{75}, denoted by $Q_{l,i}^{75}$.

(d) For every $(i, j) \in Table_{P,R}$, and for each pair (Q_l^{75}, S_m^{75}), check how many times $Q_{l,i}^{75} \oplus S_{m,j}^{75} \in \Delta_{OUT}$ is satisfied. If this number is greater than 4, apply the key recovery algorithm using Q_l^{75} as the related-key plaintext counterpart of P_0^0 and S_m^{75} as the related-key plaintext counterpart of R_0^0.

3. If the attack fails for all the 2^{32} pools of $\{S_m^{75}\}$, repeat Step 1.1 and Step 2 for another choice of P_0^0 and R_0^0.

The attack identifies the correct related-key plaintext pairs by observing the fact that if Q_l^{75} and S_m^{75} are the related-key plaintext counterparts of P_0^0 and R_0^0, respectively, then out of the 2^{20} checked pairs, it is expected that 8 satisfy Δ_{OUT}. In case that (P_0^0, Q_l^{75}) and (R_0^0, S_m^{75}) are not related-key plaintext pairs, then only 2^{-12} of the $Q_{l,i}^{75} \oplus S_{m,j}^{75}$ values are expected to satisfy Δ_{OUT}.

C.2 Analysis of the Attack

We first note that during the attack, Steps 1 and 2 are expected to be executed 2^{17} times (until encountering a pair (P_0^0, R_0^0) that satisfies the related-key differential).

Step 1 consists mainly of searching the encryption chains of P_0^0 and R_0^0, which can be efficiently performed using hash tables. Thus, Step 1 requires about 2^{21} memory accesses for finding the pairs (i, j), and another 2^{20} memory accesses for storing them.

For each pool $\{S_m^{75}\}_{m=0}^{2^{32}-1}$, steps 2(a),2(b), and 2(c) consist of generating sets of 2^{20} values 2^{33} times (2^{32} for S_m^{75}'s and 2^{32} times for Q_l^{75}'s) . Assuming that these values are generated as in the attack of Section 4.2, this stage is mainly composed of memory accesses (2^{53} for each set of S_m^{75}'s). Step 2(d) is the more complex one. By correctly indexing the tables, it is possible to try all the possible pairs of the form $(Q_{l,i}^{75}, S_{m,j}^{75})$ requiring only 2^{52} memory accesses for a pool. This is done by storing for each j all the values $S_{m,j}^{75}$ (or more precisely the 32 bits composed of the XOR of the first two words and the XOR of the last two words of $S_{m,j}^{75}$ for all m's). Then to query a specific $Q_{l,i}^{75}$ it is sufficient to compute the XOR of the first and second words, and the XOR of the third and fourth words, and to check whether this value appears in the corresponding table. Thus, we conclude that Step 2 requires 2^{54} memory accesses for any pool of S_m^{75}'s. As there are 2^{32} such pools, the total time complexity of Step 2 is 2^{86} memory accesses for each choice of the pair (P_0^0, R_0^0).

Therefore, we conclude that this attack requires 2^{103} memory accesses, which is roughly the same as the attack in Section 4.2. The data complexity of the attack is about 2^{71} related-key chosen plaintexts and ciphertexts for obtaining $Chain_Q$ and $Chain_S$. Generating the chains of all the pairs of (P_0^0, R_0^0) required for the attack is expected to require about $2^{65.5}$ related-key adaptive chosen plaintexts and ciphertexts.

Algebraic and Slide Attacks on KeeLoq

Nicolas T. Courtois[1], Gregory V. Bard[2], and David Wagner[3]

[1] University College London, Gower Street, London WC1E 6BT, UK
[2] Fordham University, NY, USA
[3] University of California - Berkeley, Berkeley CA 94720, USA

Abstract. KeeLoq is a block cipher used in wireless devices that unlock the doors and alarms in cars manufactured by Chrysler, Daewoo, Fiat, GM, Honda, Jaguar, Toyota, Volvo, Volkswagen, etc [8,9,33,34]. KeeLoq is inexpensive to implement and economical in gate count, yet according to Microchip [33] it should have "a level of security comparable to DES".

In this paper we present several distinct attacks on KeeLoq, each of them is interesting for different reasons. First we show that when about 2^{32} known plaintexts are available, KeeLoq is very weak and for example for 30 % of all keys the full key can be recovered with complexity of 2^{28} KeeLoq encryptions. Then we turn our attention to algebraic attacks with the major challenge of breaking KeeLoq given potentially a very small number of known plaintexts.

Our best "direct" algebraic attack can break up to 160 rounds of KeeLoq. Much better results are achieved in combination with slide attacks. Given about 2^{16} known plaintexts, we present a slide-algebraic attack that uses a SAT solver with the complexity equivalent to about 2^{53} KeeLoq encryptions. To the best of our knowledge, this is the first time that a full-round real-life block cipher is broken using an algebraic attack.

Keywords: block ciphers, unbalanced Feistel ciphers, slide attacks, algebraic cryptanalysis, Gröbner bases, SAT solvers, KeeLoq.

1 Introduction

KeeLoq is a lightweight block cipher designed in the 1980's and in 1995 it was sold to Microchip Technology Inc for more than 10 million US dollars as documented in [8]. Following [35], the specification of KeeLoq that can be found in [34] is "not secret" but is patented and was released only under license. In 2007, a Microchip document with the specification of KeeLoq has been made public on a Russian web site [34].

KeeLoq operates with 32-bit blocks and 64-bit keys. Compared to typical block ciphers that have a few carefully-designed rounds, this cipher has 528 extremely simple rounds. KeeLoq is not a stream cipher, it does not actually use any LFSR, and the construction only resembles an NLFSR. Therefore it is not trivial to see whether KeeLoq will be vulnerable to algebraic attacks. KeeLoq is a full-fledged "unbalanced Feistel" block cipher of compressing type,

K. Nyberg (Ed.): FSE 2008, LNCS 5086, pp. 97–115, 2008.

and we anticipate from the Luby-Rackoff theory, that such ciphers are secure if the number of rounds is sufficient. Is 528 rounds sufficient? As it turns out, KeeLoq has been designed to be fast, and requires a low number of gates to be implemented. This is quite interesting as it has been sometimes conjectured that ciphers which require a small number of gates should be vulnerable to algebraic cryptanalysis, see [19,13,18,15]. Currently algebraic attacks on block ciphers are not very powerful, for example for DES [19] and a toy cipher called CTC [17], only respectively 6 and 10 rounds can be broken, and further progress seems difficult. According to [33], KeeLoq should have "a level of security comparable to DES". In this paper we will see that the simplicity of KeeLoq makes it directly breakable by simple algebraic attacks for up to 160 rounds out of 528.

The key property that allows further more efficient attacks on KeeLoq is a sliding property: KeeLoq has a periodic structure with a period of 64 rounds. This in combination with an algebraic attack will allow us to recover the complete key for the full 528-round cipher given 2^{16} known plaintexts.

KeeLoq has unusually small block length: 32 bits. Thus, in theory the attacker can expect to recover and store the whole code-book of 2^{32} known plaintexts. Then one may wonder whether it is really useful to recover the key, as the code-book allows one to encrypt and decrypt any message. However, there are many cases in which it remains interesting for example if a master key can be recovered. We will see that given the whole code-book KeeLoq is spectacularly weak and the key will be recovered in time equivalent to about 2^{28} KeeLoq encryptions.

This paper is organised as follows: in Section 2 we describe the cipher and its usage. In Section 3, we discuss the unusual properties of block ciphers with small blocks and discuss the practical interest of key recovery attacks in this case. In Section 4 we do some preliminary analysis of KeeLoq and recall useful results about random functions. In Section 5 we describe a very fast attack that recovers the key for full KeeLoq given the knowledge of slightly less than the whole code-book. In Section 6 we demonstrate several simple algebraic attacks that work given very small quantity of known/chosen plaintexts and for a reduced number of rounds of KeeLoq. In Section 7 we study combined slide and algebraic attacks that work given about 2^{16} known plaintexts for the full 528-round cipher. In Appendix A we discuss strong keys in KeeLoq. In Appendix B we study the algebraic immunity of the Boolean function used in KeeLoq.

1.1 Notation

We will use the following notation for functional iteration:

$$f^{(n)}(x) = \underbrace{f(f(\cdots f(}_{n \text{ times}} x)\cdots))$$

2 Cipher Description

The specification of KeeLoq can be found in the Microchip product specification document [34], which actually specifies KeeLoq decryption, that can be converted to a description of the encryption, see [8,9,4]. Initially there were mistakes in [8,9] as opposed to [34,4] but they are now corrected.

The KeeLoq cipher is a strongly unbalanced Feistel construction in which the round function has one bit of output, and consequently in one round only one bit in the "state" of the cipher will be changed. Alternatively it can viewed as a modified shift register with non-linear feedback, in which the fresh bit computed by the Boolean function is XORed with one key bit.

The cipher has the total of 528 rounds, and it makes sense to view that as $528 = 512+16 = 64\times 8+16$. The encryption procedure is periodic with a period of 64 and it has been "cut" at 528 rounds, because 528 is not a multiple of 64, in order to prevent obvious slide attacks (but more advanced slide attacks remain possible as will become clear later). Let $k_{63}, \ldots, k_0$ be the key. In each round, it is bitwise rotated to the right, with wrap around. Therefore, during rounds $i, i+64, i+128, \ldots$, the key register is the same. If one imagines the 64 rounds as some $f_k(x)$, then KeeLoq is

$$E_k(x) = g_k(f_k^{(8)}(x))$$

with $g(x)$ being a 16-round final step, and $E_k(x)$ being all 528 rounds. The last "surplus" 16 rounds of the cipher use the first 16 bits of the key (by which we mean $k_{15}, \ldots, k_0$) and g_k is a functional"prefix" of f_k (which is also repeated at the end of the whole encryption process). In addition to the simplicity of the key schedule, each round of the cipher uses *only one* bit of the key. From this we see that each bit of the key is used exactly 8 times, except the first 16 bits, $k_{15}, \ldots, k_0$, which are used 9 times.

At the heart of the cipher is the non-linear function with algebraic normal form (ANF) given by:

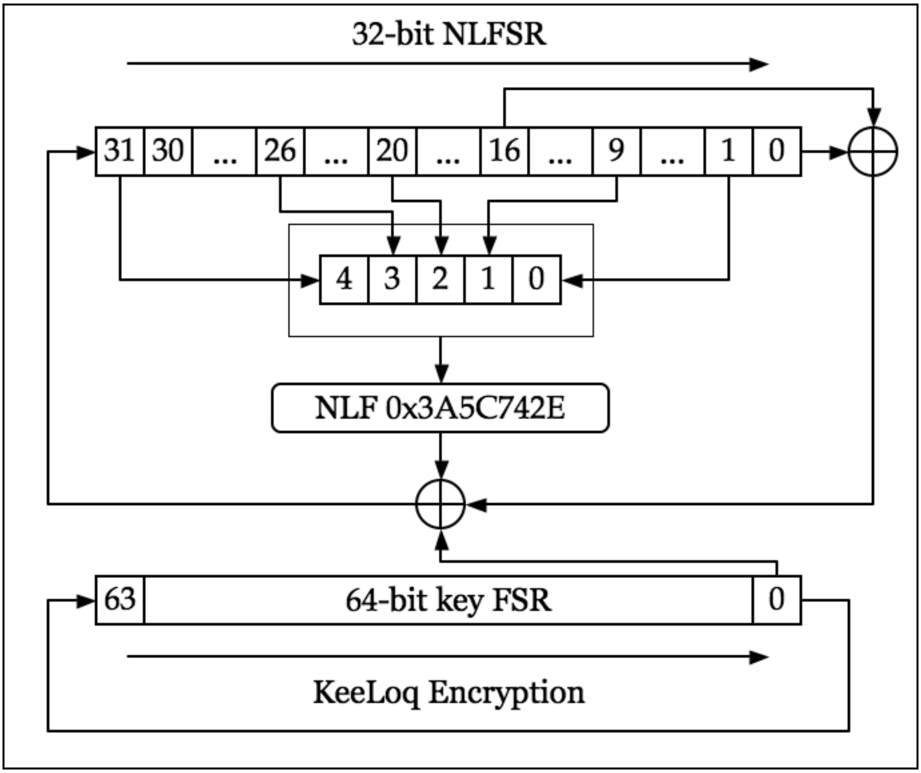
$$NLF(a,b,c,d,e) = d \oplus e \oplus ac \oplus ae \oplus bc \oplus be \oplus cd \oplus de \oplus ade \oplus ace \oplus abd \oplus abc$$

Alternatively, the specification documents available [8], say that it is "the non-linear function 3A5C742E" which means that $NLF(a,b,c,d,e)$ is equal to the i^{th} bit of that hexadecimal number, where $i = 16a + 8b + 4c + 2d + e$. For example $0,0,0,0,1$ gives $i = 1$ and the second least significant (second from the right) bit of of "3A5C742E" written in binary.

The main shift register has 32 bits, (unlike the key shift register with 64 bits), and let L_i denote the leftmost or least-significant bit at the end of round i, while denoting the initial conditions as round zero. At the end of round 528, the least significant bit is thus L_{528}, and then let $L_{529}, L_{530}, \ldots, L_{559}$ denote the 31 remaining bits of the shift register, with L_{559} being the most significant. The following equation gives the shift-register's feedback:

$$L_{i+32} = k_{i \ mod\ 64} \oplus L_i \oplus L_{i+16} \oplus NLF\left(L_{i+31}, L_{i+26}, L_{i+20}, L_{i+9}, L_{i+1}\right)$$

where $k_{63}, k_{62}, \ldots, k_1, k_0$ is the original key.



1. Initialize with the plaintext: $L_{31}, \ldots, L_0 = P_{31}, \ldots, P_0$
2. For $i = 0, \ldots, 528 - 1$ do

$$L_{i+32} = k_{i \ mod \ 64} \oplus L_i \oplus L_{i+16} \oplus NLF(L_{i+31}, L_{i+26}, L_{i+20}, L_{i+9}, L_{i+1})$$

3. The ciphertext is $C_{31}, \ldots, C_0 = L_{559}, \ldots, L_{528}$.

Fig. 1. KeeLoq Encryption

2.1 Cipher Usage

It appears that the mode in which the cipher is used depends on the car manufacturer. One possible method is a challenge-response authentication with a fixed key and a random challenge. Another popular method is to set the plaintext to 0, and increment the key at both sides. Another important mode is a so called 'hopping' or 'rolling' method described in [4,33]. In this case 16 bits of the plaintext are permanently fixed on both sides, and the attacker cannot hope get more than 2^{16} known plaintexts. More information can be found in [4,5,6].

In this paper we study the security of the KeeLoq cipher against key recovery attacks given a certain number of known or chosen plaintexts.

3 Block Ciphers with Small Blocks and Large Key Size

Most known block ciphers operate on binary strings and consist of a function $E : \{0,1\}^{\ell_K} \times \{0,1\}^{\ell_P} \rightarrow \{0,1\}^{\ell_P}$. The stereotype is that $\ell_P = \ell_K = \ell_P$, however in practical/industrial applications, this is almost never the case, for example:

- IDEA $\ell_P = 64, \ell_K = 128$.
- DES $\ell_P = 64, \ell_K = 56$.
- Two-key triple DES $\ell_P = 64, \ell_K = 112$.
- AES $\ell_P = 128, \ell_K \in \{128, 192, 256\}$.
- KeeLoq $\ell_P = 32, \ell_K = 64$.

Another stereotype is that, when $\ell_P << \ell_K$, for example in triple-DES and AES-256, the key recovery is of purely academic interest, and an attack on AES-256 that runs in time of say 2^{240} has no practical consequences. However, for cipher such as KeeLoq, the situation is different. The block size is small enough so that the whole code-book can be stored on a PC, and several key recovery attacks are feasible in practice. In what follows we will explain that, when $\ell_P << \ell_K$, and even if more or less the whole code-book is known, recovering the key can have numerous important and practical consequences.

3.1 On Key Recovery Attacks and Ciphers with Small Blocks

The *code-book* of a cipher E under a key k is defined as the set of all 2^{ℓ_P} pairs (P, C) such that $E(k, P) = C$. If $2^{\ell_P} < 2^{\ell_K}$, a natural question is why, would one want to recover the key if it is possible to have the entire code-book? There are several answers to this question depending on the circumstances:

1. If the adversary is a powerful insider, and has an oracle chosen-plaintext access to the cipher, or the whole code-book table stored in memory, the question will be of purely academic interest. Nevertheless, in real life applications, and even for such powerful attackers, the actual key recovery can be very valuable. When there is a master key in the system – it is a common practice in the industry – a successive key recovery would allow to compromise the security of the system on a much wider scale.
2. In most practical scenarios, we rather have a known-plaintext attack, and not all plaintexts will actually arise (e.g. due to padding, specific probability distribution, some values only can appear in the future, etc.). Here the adversary can recover a number of plaintext-ciphertext pairs that can be for example 60 % of all possible pairs, but typically he cannot hope to recover all pairs. Importantly, the value of pairs he doesn't have may be very large, while the value of pairs he already has can be negligible. For example, a block cipher with small block-size can be used to anonymize records in medical and financial databases. Then, key recovery would allow the adversary to have all possible pairs, some of which may potentially be valuable. A security model for ciphers with small blocks was recently studied by Granboulan and

Pornin in Section 5 of [28]: in this model, even if the adversary has the whole code-book except images of two points, his goal is to recover the images of the missing two points, which can be still very hard.

3. In many real-life situations, the code-book can be noisy, and contain errors. This can be because of transmission errors, human errors such as selecting the wrong encryption key, inadvertent interference with another system or another (active) attacker, or a defensive voluntary injection of dummy messages to frustrate the attackers. Then again, the key recovery may be the only way to know which messages were genuine.

In an extreme scenario, the whole code-book is known, but not with certainty, and a confirmation is sought. If the reader doubts the practicality of this scenario, consider the following. In 1942, the United States decrypted many messages encrypted with the famous Japanese cipher known as "Purple", forecasting an attack at "AF." Making sure that "AF" was in fact the Midway Island (which was anticipated but there was no certitude) had a pivotal impact on winning the World War II. For more details we refer to David Kahn [31].

4 Preliminary Analysis and Useful Combinatorial Facts

4.1 Preliminary Analysis of KeeLoq

Fact 4.1. Given (x, y) with $y = h_k(x)$, where h_k represents up to 32 rounds of KeeLoq, one can find the part of the key used in h_k in as much time as it takes to compute h_k.

Justification: This is because for up to 32 rounds, all state bits between round i and round $i-1$ are directly known. More precisely, after the round i, $32-i$ bits are known from the plaintext, and i bits are known from the ciphertext, for all $i = 1, 2, \ldots, 32$. Then the key bits are obtained directly: we know all the inputs of each NLF, and we know the output of it XORed with the corresponding key bit. We simply have $k_{i-32} = L_i \oplus L_{i-32} \oplus L_{i-16} \oplus NLF\left(L_{i-1}, L_{i-6}, L_{i-12}, L_{i-23}, L_{i-30}\right)$. This also shows that there will be exactly one possible key.

Remark: For more rounds it is much less simple, yet as shown in Section 6, direct algebraic attacks allow to efficiently recover the key for up to 160 rounds.

Fact 4.2. Given (x, y), one can quickly test whether it is possible that $y = g_k(x)$ for 16 rounds. The probability that a random (x, y) will pass this test is $1/2^{16}$.

Justification: After 16 rounds of KeeLoq, only 16 bits of x are changed, and 16 bits of x are just shifted. If data is properly aligned this requires a 16-bit equality test that should take only 1-2 CPU clocks.

Fact 4.3. Given (x, y) with $y = h_k(x)$, where h_k represents 48 rounds of KeeLoq, one can find all 2^{16} possible keys for h_k in as much time as 2^{16} times the time to compute h_k.

Justification: Try exhaustively all possibilities for the first 16 key bits and apply Fact 4.1.

Fact 4.4. For full KeeLoq, given a pair (p, c) with $c = E_k(p)$, it is possible to very quickly test whether p is a possible fixed point of f_k^8. All fixed points will be accepted; all but $1/2^{16}$ of the non-fixed points will be rejected.

Justification: If p is a fixed point of f^8, then $c = g_k(p)$. We simply use Fact 4.2 to test whether it is possible that $c = g_k(p)$.

4.2 Useful Facts about Fixed Points and Random Permutations

Proposition 4.1. Given a random function from n-bits to n-bits, the probability that a given point y has i pre-images is $\frac{1}{ei!}$, when $n \to \infty$.

Justification: Let y be fixed, the probability that $f(x) = y$ is $1/N$ where $N = 2^n$ is the size of the space. We get $\binom{N}{i}(1-1/N)^{N-i}1/N^i \approx \frac{1}{ei!}$ when $N \to \infty$. This is a Poisson distribution with the average number of pre-images being $\lambda = 1$.

This fact can be applied to derive statistics on the expected number of fixed points of permutations tat we encounter in cryptanalysis. In particular let $f_k(x)$ be the first 64 rounds of KeeLoq. Assuming that $f_k(x) \oplus x$ is a pseudo-random function, we look at the number of pre-images of 0 with this function. This gives immediately:

Proposition 4.2. The first 64 rounds f_k of KeeLoq have 1 or more fixed points with probability $1 - 1/e \approx 0.63$.

Proposition 4.3. Assuming f_k behaves as a random permutation, the expected number of fixed points of f_k^8 is exactly 4.

Justification: A random permutation π has 1 fixed point on average. Then in addition to possible "natural" fixed points (1 on average) the π^8 will also "inherit" all fixed points of π^2, π^4 and π^8, that for large permutations are with very high probability all distinct fixed points. A rigourous proof of this fact can be obtained from the authors, see [22].

Proposition 4.4. The probability that i) first 64 rounds f_k of KeeLoq have 1 or more fixed points, and simultaneously ii) the 512 rounds f_k^8 have $j = 4$ or more fixed points, is $e^{-1} - \frac{13}{6}e^{-15/8}$ which is about 0.29985.

Justification: Roughly, from Proposition 4.3, we expect about half of 0.63. In [22] we show how one compute such probabilities exactly with the methods of modern analytic combinatorics, see [25,38,22].

5 Attacks on KeeLoq That Use the Whole Dictionary

We will now present an attack that is extremely fast and shows that KeeLoq is very weak when (about) the whole code-book is known. In this paper we present two relatively simple versions of this attack. Additional improved versions are described in [22].

5.1 Setup and Assumptions

We assume that one can iterate through all possible 2^{32} plaintexts. This can either be obtained from a remote encryption oracle, or simply harnessing the circuitry without being able to read the key in order to clone the device. While this may sound like a practical attack scenario, it is hard to imagine a hacker patient enough to get 2^{32} known plaintext-ciphertext pairs from the device knowing that brute force is actually feasible. For simplicity we will assume that all the plaintext-ciphertext pairs are stored in a table. This requires 16 Gigabytes of RAM which is now available on a high-end PC. We also assume that the time to get one pair is about $t_r = 16$ CPU clocks. This is a realistic and conservative estimation.

In our Slide-Determine Attack we make the following assumption.

Fixed Point Assumption. We assume that there is at least one fixed point for $f_k(x)$ where $f_k(x)$ represents the first 64 rounds of the cipher. As shown in Section 4.2, this happens with probability 0.63. We recall that if x is a fixed point of $f_k(\cdot)$ than x is a fixed point of $f_k^{(8)}(\cdot)$, which are the first 512 rounds of KeeLoq. In fact several additional fixed points for f_k^8 are expected to exist, as on average f_k^8 has 4 fixed points (cf. Proposition 4.3).

The complexity of nearly all known attacks on KeeLoq greatly depends on the number of fixed points for f_k and f_k^8. In our attack that will follow, the more fixed points exist for f_k^8, the fastest can be the overall attack. The version A of our attack works for 63 % of all keys (cf. Proposition 4.2). The version B is faster but works for a smaller fraction of 30 % of all keys (this figure comes from Proposition 4.4). Other versions of this attack can be designed and are described in [22]. In contrast, for 37 % of keys for which f_k has no fixed point whatsoever, all versions of our Slide-Determine attack fail completely. In Appendix A we discuss this situation: there is a large class of "strong keys" for which the cipher is more secure.

5.2 Our Slide-Determine Attack

This attack requires 2^{32} plaintext-ciphertext pairs (p, c). We assume that (at least) one p is a fixed point of f_k. Then, slightly more than 4 of them on average are fixed points for f_k^8 (cf. Proposition 4.3). This attack occurs in three stages.

Stage 1 - Batch Guessing Fixed Points. Following our assumption there is (at least) one p is a fixed point for f_k^8. For each pair (p, c), we use Fact 4.4 to test whether it is possible that $f_k^8(p) = p$; if not, discard that pair. The complexity so far is about $t_r \cdot 2^{32}$ CPU clocks (mostly spent accessing the memory). Only about $2^{16} + 4$ pairs will survive, and all these where p is a fixed point to f_k^8.

Then following Fact 4.1 we can at the same time compute 16 bits of the key with time of about $4 \cdot 16$ CPU clocks (cf. Fact 4.1 and 6.1). To summarize, given the whole code-book and in time of about $t_r \cdot 2^{32}$ CPU clocks by assuming that p is a fixed point to f_k^8, we produce a list of about $\tau_A = 2^{16}$ triples $p, c, (k_{15}, \ldots, k_0)$. Assuming that $t_r = 16$ CPU clocks, the complexity of Stage 1 is about 2^{36} CPU clocks which is about 2^{25} KeeLoq encryptions.

Stage 1B - Filtering (Optional). This stage is optional, it is omitted in version A of our attack, and necessary in version B. We wish to be able to filter out a certain fixed proportion of these 2^{16} cases, so that the complexity of Stage 1 will dominate attack. Let j_8 be the number of fixed points for f_k^8. If $j_8 > 1$, our attack can be improved. If we omit Stage 1B, or if $j_8 = 1$ (which is quite infrequent), then the Stage 3 will dominate the attack, which as we will see later will make it noticeably slower. To bridge this gap we wish to exclude a proportion of all the 2^{16} pairs. The filtering is done as follows.

We store the triples $p, c, (k_{15}, \ldots, k_0)$ in a data structure keyed by $(k_{15}, \ldots, k_0)$. (For instance, we can have an array of size 2^{16}, where A$[i]$ points to a linked list containing all triples such that $(k_{15}, \ldots, k_0) = i$.) It allows us to count, for each value $(k_{15}, \ldots, k_0)$, the number f of triples associated with that partial-key value. In version B, we assume that $j_8 \geq 4$ which occurs for 30 % of all keys (cf. Proposition 4.4). We will then discard all triples such that $(k_{15}, \ldots, k_0)$ does not repeat 4 or more times in our list.

The worst-case complexity of Step 2 and Step 3 of our attack will be proportional to the size τ of our list (if all cases are tried). The expected size of the filtered list can be computed as follows: we assume that the keys that appear in this table are the 2^{16} outputs of a random function on 16 bits, that takes as input any of the 2^{16} pairs (p, c). Then following Proposition 4.1, the proportion of $\frac{1}{ei!}$ of keys will appear i times. The total number of keys that will appear 4 or more times is therefore equal to $2^{16} \cdot \sum_{i \geq 4} \frac{1}{ei!}$. However, we have to check all the triples which is more than all the keys. In our list of triples, each of these keys will appear i times (in some triple). In our attack, it is not merely sufficient to find a triple in our list having the correct 16 bits of the key: this is because our list contains several fixed points for f_k, but only about one fixed point for f_k^8 which is necessary to complete further stages of our attack. Accordingly, the expected number of elements to be checked (the size of our list) is $\tau_B = 2^{16} \cdot \sum_{i \geq 4} i \cdot \frac{1}{ei!} \approx 2^{12.4}$. This is the worst-case estimate for the attack version B (which works for 30 % of all keys). On average we only need about half of this number.

Stage 2 - Batch Solving. If we assume that p is a fixed point of f_k, at least one triple in our list is valid. Moreover we expect than less than 2 are valid on average, as we expect on average between 1 and 2 fixed points for f_k (we assumed there is at least one). For each surviving triple, assume that p is a fixed point, so that $c = E_k(p) = g_k(f_k^{(8)}(p)) = g_k(p)$. Note that if $f_k(p) = p$, then $p = h_k(c)$, where h_k represents the 48 rounds of KeeLoq using the last 48 key bits. Then an algebraic attack can be applied to suggest possible keys for this part. If we guess additional 16 bits of the key, such an attack with a SAT solver takes less than 0.1 s. We have found a simpler and faster method to get the same result. We use Fact 4.3 to recover 2^{16} possibilities for the last 48 key bits from the assumption that $p = h(c)$. Combined with the 16 bits pre-computed above for each triple, we get a list of at most 2^{32} possible full keys on 64 bits. We expect to compute only at most $2^{16} \cdot \tau$ of these full keys on 64 bits, before the attack succeeds. This takes time of at most $2^{16} \cdot \tau$ computations of $h_k(\cdot)$ (cf. Fact 4.3). And we have

$\tau_A = 2^{16}$ and $\tau_B = 2^{12.4}$ in versions A and B of our attack respectively. Overall Step 2 requires $2^{16} \cdot \tau \cdot 4 \cdot 48$ CPU clocks, which is approximatively $2^{28.6}$ and $2^{25.0}$ KeeLoq encryptions for respective versions A and B.

Early Abort: About half of this number is needed on average, with an early abort strategy as follows: for each of the τ triples we will execute Step 2 and Step 3. If Step 3 recovers and confirms the full key of KeeLoq, we abort the attack.

Stage 3 - Verification. Finally, we test each of these $2^{16} \cdot \tau$ complete keys (which is less or equal to 2^{32}) on one other plaintext-ciphertext pair p', c'. Most incorrect key guesses will be discarded, and only 1 or 2 keys will survive, one of them being correct. With additional few pairs we get the right key with certainty.

In version A, this stage requires up to $2^{16} \cdot \tau_A = 2^{32}$ full 528-round KeeLoq encryptions. which dominates the complexity of the whole attack. In version B, we need at most $2^{16} \cdot \tau_B$ full KeeLoq encryptions.

Complexity Analysis

The total complexity of the full attack is as follows:

Version A: The worst-case complexities of stages 1, 2 and 3 are 2^{25}, $2^{28.6}$ and $2^{32.0}$ KeeLoq encryptions respectively. The total is is $2^{32.1}$ KeeLoq encryptions.

On average, with early abort of Stages 2 and 3, after trying on average half of τ_A triples, as described above (Step 1 has to be executed in entirety), we get an average complexity of about $2^{31.1}$ KeeLoq encryptions.

Version B: In this version, the worst-case complexities of stages 1, 2 and 3 are 2^{25}, 2^{25} and $2^{28.4}$ KeeLoq encryptions. The total is $2^{28.7}$ KeeLoq encryptions.

On average, we get about $2^{27.7}$ KeeLoq encryptions.

Summary of Our Slide-Determine Attack. To summarize, for 30 % of all keys our Slide-Determine Attack version B allows to recover the key in average time equivalent to about 2^{28} KeeLoq encryptions. Overall, for all 63 % of keys our Slide-Determine Attack version A requires about 2^{31} KeeLoq encryptions on average. No attack of comparable efficiency is known for the remaining 37 % of keys.

6 Direct Algebraic Attacks on KeeLoq

Our goal is to recover the key of the cipher by solving a system of multivariate equations given a small quantity of known, chosen or random plaintexts, as in [13]. Very few such attacks are really efficient on block ciphers. For example DES can be broken for up to 6 rounds by such attacks, see [19]. For KeeLoq, due to its simplicity, up to 160 rounds can be directly attacked, without (not yet) exploiting the sliding properties of the cipher. This is in particular interesting for the 37 % of keys for which all our sliding attacks fail.

6.1 How to Write the Equations

We write equations in a straightforward way: namely by following directly the description of Fig 1. One new variable represents the output of the NLF in the current round. In addition, in order to decrease the degree, we add two additional variables per round, to represent the monomials $\alpha = ab$ and $\beta = ae$, and add equations of the form $\alpha_i = a_i b_i$ and $\beta_i = a_i e_i$.

This means we have:

$$y = NLF(a, b, c, d, e) = d \oplus e \oplus ac \oplus \beta \oplus bc \oplus be \oplus cd \oplus de \oplus \beta d \oplus \beta c \oplus \alpha d \oplus \alpha c$$

which permits us to write

$$\begin{aligned} L_{i+32} &= k_{i \bmod 64} \oplus L_i \oplus L_{i+16} \oplus L_{i+9} \oplus L_{i+1} \\ &\oplus L_{i+31}L_{i+20} \oplus \beta_i \oplus L_{i+26}L_{i+20} \oplus L_{i+26}L_{i+1} \oplus L_{i+20}L_{i+9} \\ &\oplus L_{i+9}L_{i+1} \oplus \beta_i L_{i+9} \oplus \beta_i L_{i+20} \oplus \alpha_i L_{i+9} \oplus \alpha_i L_{i+20} \\ \alpha_i &= L_{i+31}L_{i+26} \\ \beta_i &= L_{i+31}L_{i+1} \end{aligned}$$

These three equations need merely be repeated for each round.

The values of the plaintext, the ciphertext, and a certain number of key bits that we may fix (i.e. guess, cf. Section 6.2) during the attack are written as separate equations (for example we write that $L_31 = 1$ for the leftmost bit of the plaintext). Thus, given r rounds of the cipher, and for each known plaintext, assuming that F bits of the key are known, we will get a system of $3r+32+32+F$ multivariate quadratic equations with $3r + 64 + 32$ variables: these are all the $L_0, \ldots, L_{r+31}$, the key variables k_i, the α_i and the β_i. Out of these the values of $32 + 32 + F$ variables are already known. It should be noted that these equation and monomial counts are exact, and that this system is overdefined. The total number of distinct monomials that appear in these equations is roughly $12r$.

The equations are written for one or several known plaintexts. This will be our known-plaintext attack. In another version, we consider that the cipher is used in the counter mode, i.e. the set of plaintexts forms a set of consecutive integers encoded on 32 bits. This will be called a counter mode attack. Several complete and working examples of equations can be downloaded from [10].

6.2 Direct Algebraic Attacks on KeeLoq Vs. Brute Force

The equations of KeeLoq are of very low degree (i.e. 2), and very sparse. One can try to solve with an off-the-shelf computer algebra system such as Magma's implementation of F4 algorithm [24] or Singular's slimgb() algorithm [39]. We have also tried a much simpler method called ElimLin and described in [19]. Another family of techniques are SAT solvers. Any system of multivariate equations is amenable for transformation into a CNF-SAT problem, using the methods of [20].

Fact 6.1. An optimised assembly language implementation of r rounds of KeeLoq is expected to take only about $4r$ CPU clocks.

Justification: See footnote 4 in [4].

Thus, the complexity of an attack on r rounds of KeeLoq with k bits of the key should be compared to $4r \times 2^{k-1}$ which is the expected complexity of the brute force key search. For example, for full KeeLoq, the reference complexity for the exhaustive key search is about 2^{75} CPU clocks. Assuming that the CPU runs at 2.5 GHz, one can execute about 2^{43} CPU clocks per hour. Consider the following example. Suppose we guess 32 key bits for example $k_1 = 0, k_2 = 1, \ldots$. Suppose that the remaining key bits are found on a PC in less than an hour, or $< 2^{43}$ CPU clocks. In reality, the attacker is not given 32 bits of the key. Instead one can guess them and on average 2^{31} such guesses must be made. With early abort of unsuccessful tries after for example 1.5 hours, the expected running time is $< 2^{43}2^{31+1}$ or $< 2^{75}$, which is faster than brute force.

Note: In the real life hackers recover the KeeLoq key by brute force with FPGAs which takes about two weeks, see [8].

6.3 Frontal Assault – Elimination and Gröbner Bases Attacks

Example 1. For example, we consider 64 rounds of KeeLoq and 2 known plaintexts, and we run ElimLin as described in [19]. In 5 seconds, the program manages to eliminate all but 130 variables out of the initial 512 variables. Moreover, in the linear span of the equations after ElimLin, the program is able to find one equation of degree 2, that involves *only* the 64 key variables and in which all the internal variables of the cipher are eliminated. This is sufficient to show that 64 rounds are very easy to break by Gröbner bases. For example, we may proceed as follows: for each new pair of known plaintexts, we get a new equation of this type. Given a sufficient number of known plaintexts (a small multiple of 64 will be sufficient), we will get a very overdefined system of equations with 64 variables. Such systems are known to be easily solvable by the XL algorithm and Gröbner bases, see [12,11,1].

Example 2. Here also, we consider 64 rounds of KeeLoq and 4 known plaintexts, and we run ElimLin as described in [19]. We fix 10 key bits to their true values. Then the remaining 54 key bits are recovered by ElimLin alone in 8 seconds. With Singular slimgb() function [39] the same computation takes 1 minute.

Example 3. With 64 rounds, 2 plaintexts that differ only in 1 bit, (it is no longer a known plaintext attack, but rather a chosen plaintext attack), and with 14 key bits fixed, the key is computed by ElimLin in 7 seconds and by Singular in 10 seconds.

Example 4. With 128 rounds and 128 plaintexts in the counter mode (the plaintexts are consecutive integers on 32-bits), and 30 bits fixed, the remaining 34 bits are recovered by ElimLin in 3 hours. This is slightly faster than brute force.

6.4 Cryptanalysis of KeeLoq with SAT Solvers

From [19], one may expect that better results will be obtained with SAT solvers. Given some number of pairs of plaintext and ciphertexts, over the whole 528 rounds, we rewrite the equations as a SAT problem and try to solve them. We write equations as polynomials (cf. previous section) and use the simplest version of the ANF to CNF conversion method described in [20].

Example 5. For full 528 rounds of KeeLoq, these attacks remain much slower than exhaustive search. For example with 2 plaintexts in counter mode (two consecutive integers on 32-bits) and 48 bits fixed, the remaining 16 key bits are recovered in 30 seconds with our conversion to CNF and MiniSat 2.0., done as described in [19,20]. This is much slower than brute force. However, with a reduced number of rounds, the results are more interesting.

Example 6. For 64 rounds of KeeLoq and 2 known plaintexts, the key is recovered by MiniSat 2.0. in 0.3 s.

Example 7. For 96 rounds of KeeLoq, 4 known plaintexts, and when 20 key bits are guessed, the key is recovered by MiniSat 2.0. in 0.4 s.

Example 8. With 128 rounds and 2 known plaintexts, and 30 bits guessed, the remaining 34 bits are recovered in 150 s by MiniSat 2.0. This is about 80x faster than brute force.

Example 9. With 160 rounds, 2 plaintexts in counter mode, and 30 bits guessed, the remaining 34 bits are recovered in 233 s by MiniSat 2.0. This is clearly faster than brute force.

We note that the maximum number of rounds that we can break faster than by exhaustive search by our best algebraic attack with SAT solvers is 160 rounds. This attack does not exploit the periodicity of the cipher and uses an extremely low number of know plaintext-ciphertext pairs. In comparison up to 32 rounds can be broken directly "by hand" (cf. Fact 4.1). We also note that when the number of rounds is reduced to 64, the full key can be obtained almost instantly. This fact gave inspiration to design Slide-Algebraic attacks.

7 Combining Slide and Algebraic Attacks on KeeLoq

If the number of rounds were 512, and not 528, then it would be easy to analyse KeeLoq as an 8-fold iteration of 64 rounds. The last 16 rounds are a "barrier", which we can remove by guessing the 16 bits of the key used in those 16 rounds. These are the first 16 key bits, or $k_0, \ldots, k_{15}$, and the guess is correct with probability 2^{-16}. This is what we will do in our Slide-Determine Attack and our Slide-Algebraic Attack 1. Alternatively (as we will see in our Slide-Algebraic Attack 2), we may assume/guess some particular property of the 512 rounds of the cipher and try to recover the 16 (or more) bits that confirm this property.

Classical sliding attacks [3,26,30] exploit pairs of plaintext that have the following property:

Definition 7.1. Given a block cipher with periodic structure of the form $E_k(x) = g_k(f_k^{(m)}(x))$, $m > 1$, we call a "slid pair" any pair of plaintexts (P_i, P_j) such that $f_k(P_i) = P_j$.

7.1 Slide-Algebraic Attack 1

A simple sliding attack on KeeLoq would proceed as follows.

1. We guess the 16 key bits of g_k which gives us "oracle access" to 512 rounds of KeeLoq that we denote by $O = f_k^{(8)}$.
2. We consider 2^{16} known plaintexts (P_i, C_i).
3. By birthday paradox, one pair (P_i, P_j) is a "slid pair" for 64 rounds.
4. From this, one can derive as many known plaintexts for 64 rounds of KeeLoq as desired. For example, if $f_k(P_i) = P_j$ then $f_k(O(P_i)) = O(P_j)$. Additional "slid pairs" are obtained by iterating O twice, three times etc..
5. The whole attack has to be run about 2^{32} times, to find the correct "slid pair" (P_i, P_j).

In all with guessing the key of g_k there are 2^{48} possibilities to check. For each potential value for the first 16 bits of the key, and for each couple (P_i, P_j) we compute some 4 plaintext-ciphertext pairs for 64 rounds and then the key is recovered by MiniSat (cf. above) in 0.4 s which is about 2^{30} CPU clocks. The total complexity of the attack is about 2^{78} CPU clocks which is more than the exhaustive search.

7.2 Slide-Algebraic Attack 2

Another, better sliding attack proceeds as follows.

1. We do *not* guess 16 key bits, they will be determined later.
2. We consider 2^{16} known plaintexts (P_i, C_i).
3. By birthday paradox, one pair (P_i, P_j) is a "slid pair": $f_k(P_i) = P_j$.
4. Then the pair (C_i, C_j) is a plaintext-ciphertext pair for a "slided" version of the same cipher: starting at round 16 and finishing before round 80. This is to say a cipher with absolutely identical equations in every respect except for the (permuted) subscripts of the k_i.
5. From the point of view of multivariate equations and algebraic cryptanalysis, this situation is **not** much different than in Example 6 above solved in 0.3 seconds. We have one system of equations with the pair (P_i, P_j) for the first 64 rounds, and the same system of equations with the pair (C_i, C_j) and the key bits that are rotated by 16 positions.
6. We did write this system of equations and try ElimLin and MiniSat. For example with 15 first bits of the key fixed, ElimLin solves the system in 8 seconds. Better results are obtained with MiniSat, and without guessing any key variables, the key is computed in typically[1] about 2 seconds. Thus, with ElimLin, we can recover the key in about 2^{49} CPU clocks, and with MiniSat, we can do it in about 2^{32} CPU clocks.
7. There are about 2^{32} pairs (P_i, P_j) to be tried.

[1] We have written these equations for 10 different random keys with randomly chosen plaintexts, and the timings we obtained were: 2.3, 9.1, 1.5, 0.5, 4.4, 0.3, 8.1, 1.8, 0.4, 0.6 seconds. Here the average time is 2.9 s and the median time is 1.65 s.

The total complexity of the attack, in the version with MiniSat is about $2^{32+32} = 2^{64}$ CPU clocks which is much faster than exhaustive search that requires about 2^{75} CPU clocks.

Summary. Our Slide-Algebraic Attack 2 can break KeeLoq within 2^{64} CPU clocks given 2^{16} known plaintexts. This is about 2^{53} KeeLoq encryptions. The attack is realistic, practical and has been fully implemented.

8 Conclusions

In this paper we described several key recovery attacks on KeeLoq, a block cipher with a very small block size and a simple periodic structure. KeeLoq is widespread in the automobile industry and is used by millions of people every day. Recently it has been shown that for a more complex cipher such as DES, up to 6 rounds can be broken by an algebraic attack given only one known plaintext [19]. In this paper we showed that up to 160 rounds of KeeLoq can be broken directly using MiniSat 2.0. algorithm with only 2 chosen plaintexts.

In combination with sliding attacks, an algebraic attack one the full 528-round KeeLoq is possible. Given about 2^{16} known plaintexts, we have proposed a working slide-algebraic attack equivalent to 2^{53} KeeLoq encryptions. In particular, in the so called 'hopping' or 'rolling' mode described in [4,5,6,33], one cannot obtain more than 2^{16} known plaintexts. We are the first to have proposed such an attack on KeeLoq (all previous attacks required 2^{32} known plaintexts). Our attack is practical and was implemented with little programming effort.

We also showed that if as many as 2^{32} known plaintexts are available, KeeLoq is in fact extremely weak. For example, for 30 % of all keys, we can recover the key of the full cipher with complexity equivalent to 2^{28} KeeLoq encryptions. This

Table 1. Comparison of our attacks to other attacks reported on KeeLoq

Type of attack	Data	Time	Memory	Reference
Pure Algebraic/SAT	2 KP	2^{73}	small	Our Example 5
Brute Force	2 KP	2^{63}	small	
Slide-Algebraic	2^{16}KP	2^{67}	small	Our Slide-Algebraic Attack 1
Slide-Algebraic	2^{16}KP	$\mathbf{2^{53}}$	small	Our Slide-Algebraic Attack 2
Slide-Meet-in-the-Middle	2^{16}KP	$\mathbf{2^{46}}$	small	Biham, Dunkelman *et al*[23]
Slide-Meet-in-the-Middle	2^{16}CP	$\mathbf{2^{45}}$	small	Biham, Dunkelman *et al*[23]
Slide-Correlation	2^{32}KP	2^{51}	16 Gb	Bogdanov[4,5]
Slide-Cycle-Algebraic	2^{32}KP	2^{40}	18 Gb	Attack 3 in [21]
Slide-Cycle-Correlation	2^{32}KP	2^{40}	18 Gb	Bogdanov [5]
				Two versions:
Slide-Determine	2^{32}KP	$\mathbf{2^{31}}$	16 Gb	A: for 63 % of all keys
Slide-Determine	2^{32}KP	$\mathbf{2^{28}}$	16 Gb	B: for 30 % of all keys

Legend: The unit of time complexity here is one KeeLoq encryption.

attack can be prevented by a class of "strong keys" we defined that decreases the effective key space from 64 bits to 62.56 bits.

KeeLoq is a weak and simple cipher, and has several vulnerabilities. It is interesting to note that attacks that use sliding properties can be quite powerful because typically (in all our Slide-Determine and Slide-Algebraic Attacks) their complexity simply does not depend on the number of rounds of the cipher.

The results of this paper can be compared to [4,5,6], other very recent work on KeeLoq. Recently, another attack with 2^{16} KP and time about 2^{45} KeeLoq encryptions was proposed by Biham, Dunkelman *et al.* [23]. Knowing which is the fastest attack on one specific cipher, and whether one can really break into cars and how, should be secondary questions in a scientific paper. Instead, in cryptanalysis we need to study a variety of attacks on a variety of ciphers. Brute force will be in fact maybe the only attack that will be executed in practice by hackers. It is precisely by attacking weak ciphers such as KeeLoq in many different ways that we discover many interesting attacks, and some important attacks such as algebraic attacks would never be discovered without extensive experimentation.

Acknowledgments. We thank Sebastiaan Indesteege and Sean O'Neil for valuable help.

References

1. Bardet, M., Faugère, J.-C., Salvy, B.: On the complexity of Gröbner basis computation of semi-regular overdetermined algebraic equations. In: Proceedings of International Conference on Polynomial System Solving (ICPSS, Paris, France), pp. 71–75 (2004)
2. Biryukov, A., Wagner, D.: Advanced Slide Attacks. In: Preneel, B. (ed.) EUROCRYPT 2000. LNCS, vol. 1807, pp. 589–606. Springer, Heidelberg (2000)
3. Biryukov, A., Wagner, D.: Slide Attacks. In: Knudsen, L.R. (ed.) FSE 1999. LNCS, vol. 1636, pp. 245–259. Springer, Heidelberg (1999)
4. Bogdanov, A.: Cryptanalysis of the KeeLoq block cipher, http://eprint.iacr.org/2007/055
5. Bogdanov, A.: Attacks on the KeeLoq Block Cipher and Authentication Systems. In: 3rd Conference on RFID Security 2007, RFIDSec (2007)
6. Bogdanov, A.: Linear Slide Attacks on the KeeLoq Block Cipher. In: The 3rd SKLOIS Conference on Information Security and Cryptology (Inscrypt 2007). LNCS. Springer, Heidelberg (2007)
7. Cid, C., Babbage, S., Pramstaller, N., Raddum, H.: An Analysis of the Hermes8 Stream Cipher. In: Pieprzyk, J., Ghodosi, H., Dawson, E. (eds.) ACISP 2007. LNCS, vol. 4586, pp. 1–10. Springer, Heidelberg (2007)
8. Keeloq wikipedia article. On 25 January 2007 the specification given here was incorrect and was updated since, http://en.wikipedia.org/wiki/KeeLoq
9. Keeloq C source code by Ruptor, http://cryptolib.com/ciphers/
10. Courtois, N.: Examples of equations generated for experiments with algebraic cryptanalysis of KeeLoq, http://www.cryptosystem.net/aes/toyciphers.html
11. Courtois, N., Patarin, J.: About the XL Algorithm over GF(2). In: Joye, M. (ed.) CT-RSA 2003. LNCS, vol. 2612, pp. 141–157. Springer, Heidelberg (2003)

12. Courtois, N., Shamir, A., Patarin, J., Klimov, A.: Efficient Algorithms for solving Overdefined Systems of Multivariate Polynomial Equations. In: Preneel, B. (ed.) EUROCRYPT 2000. LNCS, vol. 1807, pp. 392–407. Springer, Heidelberg (2000)
13. Courtois, N., Pieprzyk, J.: Cryptanalysis of Block Ciphers with Overdefined Systems of Equations. In: Zheng, Y. (ed.) ASIACRYPT 2002. LNCS, vol. 2501, pp. 267–287. Springer, Heidelberg (2002)
14. Courtois, N., Meier, W.: Algebraic Attacks on Stream Ciphers with Linear Feedback. In: Biham, E. (ed.) EUROCRYPT 2003. LNCS, vol. 2656, pp. 345–359. Springer, Heidelberg (2003)
15. Courtois, N.: General Principles of Algebraic Attacks and New Design Criteria for Components of Symmetric Ciphers. In: Dobbertin, H., Rijmen, V., Sowa, A. (eds.) AES 2005. LNCS, vol. 3373, pp. 67–83. Springer, Heidelberg (2005)
16. Courtois, N.: The Inverse S-box, Non-linear Polynomial Relations and Cryptanalysis of Block Ciphers. In: Dobbertin, H., Rijmen, V., Sowa, A. (eds.) AES 2005. LNCS, vol. 3373, pp. 170–188. Springer, Heidelberg (2005)
17. Courtois, N.T.: How Fast can be Algebraic Attacks on Block Ciphers? In: Biham, E., Handschuh, H., Lucks, S., Rijmen, V. (eds.) online proceedings of Dagstuhl Seminar 07021, Symmetric Cryptography (January 07-12, 2007), http://drops.dagstuhl.de/portals/index.php?semnr=07021, http://eprint.iacr.org/2006/168/ ISSN 1862 - 4405
18. Courtois, N.: CTC2 and Fast Algebraic Attacks on Block Ciphers Revisited, http://eprint.iacr.org/2007/152/
19. Courtois, N., Bard, G.V.: Algebraic Cryptanalysis of the Data Encryption Standard. In: Cryptography and Coding, 11-th IMA Conference, Cirencester, UK, December 18-20, 2007. Springer, Heidelberg (2007), eprint.iacr.org/2006/402/; Also presented at ECRYPT workshop Tools for Cryptanalysis, Krakow, September 24-25 (2007)
20. Bard, G.V., Courtois, N.T., Jefferson, C.: Efficient Methods for Conversion and Solution of Sparse Systems of Low-Degree Multivariate Polynomials over GF(2) via SAT-Solvers, http://eprint.iacr.org/2007/024/
21. Courtois, N., Bard, G.V., Wagner, D.: Algebraic and Slide Attacks on KeeLoq, Older preprint with using incorrect specification of KeeLoq, eprint.iacr.org/2007/062/
22. Courtois, N., Bard, G.V., Wagner, D.: An Improved Algebraic-Slide Attack on KeeLoq, A sequel to the oresent paper (preprint available from the authors)
23. Biham, E., Dunkelman, O., Indesteege, S., Keller, N., Preneel, B.: How to Steal Cars – A Practical Attack on KeeLoq, Crypto 2007, rump session talk (2007); Full paper will be presented at Eurocrypt 2008 and published in Springer LNCS, http://www.cosic.esat.kuleuven.be/keeloq/keeloq-rump.pdf
24. Faugère, J.-C.: A new efficient algorithm for computing Gröbner bases (F_4). Journal of Pure and Applied Algebra 139, 61–88 (1999), www.elsevier.com/locate/jpaa
25. Flajolet, P., Sedgewick, R.: Analytic Combinatorics, 807 pages. Cambridge University Press, Cambridge (to appear, 2008), http://algo.inria.fr/flajolet/Publications/book.pdf
26. Phan, R.C.-W., Furuya, S.: Sliding Properties of the DES Key Schedule and Potential Extensions to the Slide Attacks. In: Lee, P.J., Lim, C.H. (eds.) ICISC 2002. LNCS, vol. 2587, pp. 138–148. Springer, Heidelberg (2003)
27. Furuya, S.: Slide Attacks with a Known-Plaintext Cryptanalysis. In: Kim, K.-c. (ed.) ICISC 2001. LNCS, vol. 2288, pp. 214–225. Springer, Heidelberg (2002)

28. Granboulan, L., Pornin, T.: Perfect Block Ciphers with Small Blocks. In: Biryukov, A. (ed.) FSE 2007. LNCS, vol. 4593, pp. 452–465. Springer, Heidelberg (2007)
29. Gemplus Combats SIM Card Cloning with Strong Key Security Solution, Press release, Paris (November 5, 2002), http://www.gemalto.com/press/gemplus/2002/r_d/strong_key_05112002.htm
30. Grossman, E.K., Tuckerman, B.: Analysis of a Feistel-like cipher weakened by having no rotating key, IBM Thomas J. Watson Research Report RC 6375 (1977)
31. Kahn, D.: The Codebreakers, The Comprehensive History of Secret Communication from Ancient Times to the Internet (first published in 1967) (new chapter added in 1996)
32. Marraro, L., Massacci, F.: Towards the Formal Verification of Ciphers: Logical Cryptanalysis of DES. In: Proc. Third LICS Workshop on Formal Methods and Security Protocols, Federated Logic Conferences (FLOC 1999) (1999)
33. Microchip. An Introduction to KeeLoq Code Hopping (1996), http://ww1.microchip.com/downloads/en/AppNotes/91002a.pdf
34. Microchip. Hopping Code Decoder using a PIC16C56, AN642 (1998), http://www.keeloq.boom.ru/decryption.pdf
35. Microchip. Using KeeLoq to Validate Subsystem Compatibility, AN827 (2002), http://ww1.microchip.com/downloads/en/AppNotes/00827a.pdf
36. MiniSat 2.0. An open-source SAT solver package, by Niklas Eén, Niklas Sörensson, http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/
37. Mironov, I., Zhang, L.: Applications of SAT Solvers to Cryptanalysis of Hash Functions. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 102–115. Springer, Heidelberg (2006), http://eprint.iacr.org/2006/254
38. Riedel, M.R.: Random Permutation Statistics, http://www.geocities.com/markoriedelde/papers/randperms.pdf
39. Singular, A.: Free Computer Algebra System for polynomial computations, http://www.singular.uni-kl.de/

A Strong Keys in KeeLoq

It is possible to see that the manufacturer or the programmer of a device that contains KeeLoq can check each potential key for fixed points for f_k. If it has any, that key can be declared “weak” and never used. This means that 63% of keys will be weak, and changes the effective key space from 64 bits to 62.56 bits, which is in fact a small loss. This appears to be practical for KeeLoq because the size of the plaintext-space is only 2^{32} and can be checked. A similar strong-key solution was in 2002 patented and commercialized by Gemplus corporation (currently Gemalto) to prevent GSM SIM cards from being cloned, see [29]. This removes our fastest attack on KeeLoq, Slide-Determine Attack. Further research is needed to see what is the best attack on KeeLoq in this case, and whether it is also necessary to remove fixed points for $f_k^{(2)}$.

B Algebraic Immunity and Boolean Function Used in KeeLoq

The security of KeeLoq depends on the quality of KeeLoq Boolean function NLF. We have:

$$y = NLF(a, b, c, d, e) = d \oplus e \oplus ac \oplus ae \oplus bc \oplus be \oplus cd \oplus de \oplus ade \oplus ace \oplus abd \oplus abc$$

Following [4] , this function is weak with respect to correlation attacks, it is 1-resilient but it is not 2-resilient and can in fact be quite well approximated by the linear function $d \oplus e$.

From the point of view of algebraic cryptanalysis, the fundamental question to consider is to determine the "Algebraic Immunity" of the NLF, which is also known "Graph Algebraic Immunity" or "I/O degree". We found that it is only 2, and one can verify that this NLF allows one to write the following I/O equation of degree 2 with no extra variables:

$$(e + b + a + y) * (c + d + y) = 0$$

However, there is only 1 such equation, and this equation by itself does *not* give a lot of information on the NLF of KeeLoq. This equation is "naturally" true with probability 3/4 whatever is the actual NLF used. It is therefore easy to see that this equation alone does **not** fully specify the NLF, and taken alone cannot be used in algebraic cryptanalysis. When used in combination with other equations, this should allow some algebraic attacks to be faster, at least slightly. At present time we are not aware of any concrete attack on KeeLoq that is enabled or aided by using this equation.

A Meet-in-the-Middle Attack on 8-Round AES

Hüseyin Demirci[1] and Ali Aydın Selçuk[2]

[1] Tübitak UEKAE, 41470 Gebze, Kocaeli, Turkey
huseyind@uekae.tubitak.gov.tr
[2] Department of Computer Engineering
Bilkent University, 06800, Ankara, Turkey
selcuk@cs.bilkent.edu.tr

Abstract. We present a 5-round distinguisher for AES. We exploit this distinguisher to develop a meet-in-the-middle attack on 7 rounds of AES-192 and 8 rounds of AES-256. We also give a time-memory tradeoff generalization of the basic attack which gives a better balancing between different costs of the attack. As an additional note, we state a new square-like property of the AES algorithm.

Keywords: AES, Rijndael, meet-in-the-middle cryptanalysis, square attack.

1 Introduction

In year 2000, the Rijndael block cipher was adopted by NIST as the Advanced Encryption Standard (AES), the new standard encryption algorithm of the US government to replace DES. The algorithm is a member of the family of square-type algorithms [7] designed by Vincent Rijmen and John Daemen. It is currently one of the most widely used and analyzed ciphers in the world.

AES is a 128-bit block cipher and accepts key sizes of 128, 192 and 256 bits. These versions of AES are called AES-128, AES-192 and AES-256 and the number of rounds for these versions are 10, 12 and 14 respectively. The algorithm is easy to understand, but the underlying mathematical ideas are strong. It has an SP-network structure. Interaction between the operations is chosen so that it satisfies full diffusion after two rounds. There is only one non-linear function in the algorithm, but it does not seem to have any considerable weakness so far.

AES has been remarkably secure against attacks. Some related key attacks can go up to 10 rounds on AES-192 and AES-256 with a complexity close to the complexity of exhaustive search. Attacks that are not of related-key type have been unable to go any further than 8 rounds. Most successful attacks in this class have been based on the square property observed by the designers of the Square algorithm [7].

In this paper we provide a distinguisher on 5 inner rounds of AES. This distinguisher relates a table entry of the fifth round to a table entry of the first round using 25 parameters that remain fixed throughout the attack. Using this distinguisher, we are able to attack up to 8 rounds of AES-256. For attacking

K. Nyberg (Ed.): FSE 2008, LNCS 5086, pp. 116–126, 2008.

AES-192, we use a birthday-paradox-like approach to reduce the precomputation complexity, which enables a 7-round attack on AES-192. Our attack is also related to the meet-in-the-middle attack of Demirci et al. [9] on the IDEA block cipher, where a large sieving set is precomputed according to a certain distinguishing property of the cipher, and this set is later used to discover the round keys by a partial decryption.

This paper proceeds as follows: In Section 2 we briefly explain the AES block cipher and give a survey of the previous attacks. In Section 3, we review the 4-round AES distinguisher of Gilbert and Minier [12]. In Section 4, we introduce our 5-round distinguisher for AES. In Section 5, we describe our attacks on AES-192 and AES-256 based on this distinguisher. We conclude the paper with a summary of the results in Section 6. As an additional note, we present a novel square-like property of the AES algorithm in the appendix.

2 The AES Encryption Algorithm

The AES encryption algorithm organizes the plaintext as a 4×4 table of 1-byte entries, where the bytes are treated as elements of the finite field $GF(2^8)$. There are three main operations used in AES: the s-box substitution, shift row, and mix column operations. There is a single s-box substitution used for all entries of the table based on the inverse mapping in $GF(2^8)$ plus an affine mapping, which is known to have excellent differential and linear properties [19]. The shift row operation shifts the ith row i units left for $i = 0, 1, 2, 3$. Mix column operation is an MDS matrix multiplication which confuses the four entries of each column of the table. Key mixing is done at the end of each round where the bytes of the round key are XORed to the corresponding plaintext bytes of the table. Initially there is a key whitening before encryption begins, and in the final round there is no mix column operation. The key scheduling of AES is almost linear. Our analysis is independent of the key schedule algorithm. The interaction between the operations is designed in such a way that full diffusion is obtained after two rounds. Full details of the encryption, decryption, and key schedule algorithms can be found in [11].

AES has been remarkably resistant against attacks. Although different attacks have been tried on reduced-round versions, there is no way to break the actual cipher faster than exhaustive search. The algorithm designers applied the square attack [7] to the cipher. The attack uses about 2^{32} chosen plaintexts and breaks 6 rounds of AES with about 2^{72} complexity. The square attack has been improved [10] and the workload has been reduced to 2^{46}. For the key lengths 192 and 256 bits, the attack can be increased one more round with the help of the key schedule [18]. In [12] a collision attack has been applied to the cipher using a distinguishing property of the four-round encryption. With 2^{32} chosen plaintexts, the attack breaks 7 rounds of AES-192 and AES-256 with a complexity of 2^{140}. For AES-128, the attack is marginally faster than exhaustive search. The impossible differential attack has been applied up to 7 rounds of AES [3,5,22,20,21]; but the complexities of these attacks are higher than the square attack. Biryukov applied the boomerang attack technique to 5 and 6 rounds of the cipher [4]. For the 128 bit key length, the boomerang attack breaks 5 rounds of AES using 2^{46} adaptive chosen plaintexts in 2^{46} steps of analysis. The 6-round boomerang attack requires 2^{78} chosen plaintexts, 2^{78} steps of analysis, and 2^{36} bytes of memory. There is also a class of algebraic attacks applied on AES [6]. The authors write the AES S-box as a system of implicit quadratic equations. As a result, the cryptanalysis of the system turns out to be solving a huge system of quadratic equations. In [6], XSL method is suggested if the system of equations is overdefined and sparse which is the case for AES. Recently, related key attacks have been applied to the cipher [1,2,15,14,17,23]. These attacks work up to 10 rounds of AES-192 and AES-256.

Throughout the paper, we use $K^{(r)}$ and $C^{(r)}$ to denote the round key and the ciphertext of the rth round; $K_{ij}^{(r)}$ and $C_{ij}^{(r)}$ denote the byte values at row i, column j. The arithmetic among table entries are in $GF(2^8)$, where addition is the same as bit-wise XOR. By a one round AES encryption, we mean an inner round without whitening or exclusion of the mixcolumn operation unless otherwise stated.

2.1 The Square Property

The square attack [7] is the first attack on AES, which was applied by the designers of the algorithm [8]. Proposition 1 states the distinguishing property the square attack exploits.

Throughout this paper by an active entry, we mean an entry that takes all byte values between 0 and 255 exactly once over a given set of plaintexts. By a passive entry we mean an entry that is fixed to a constant byte value.

Proposition 1 ([8]). *Take a set of 256 plaintexts so that one entry in the plaintext table is active and all the other entries are passive. After applying three rounds of AES, the sum of each entry over the 256 ciphertexts is 0.*

This property leads to a straightforward attack on 4 rounds of AES where the last round key is searched and decrypted and the third round outputs are checked for this property. This attack can be extended one round from the top and one round from the bottom so that 6 rounds of AES can be attacked using this property [7,10].

The idea behind the square attack still forms the basis of most of the analysis on AES. Therefore, obtaining more square-like properties of the cipher is essential for evaluating its security. We state such a new square-like property of the AES algorithm in the appendix.

3 A 4-Round Distinguisher of AES

In [12], Gilbert and Minier showed an interesting distinguishing property for 4 rounds of AES: Consider the evolution of the plaintext over 4 inner rounds, with no whitening. Let a_{ij} denote the ith row, jth column of the plaintext. After the first s-box transformation, define $t_{ij} = S(a_{ij})$. At the end of round 1, our state matrix is of the form:

$2t_{11} + c_1$	m_{12}	m_{13}	m_{14}
$t_{11} + c_2$	m_{22}	m_{23}	m_{24}
$t_{11} + c_3$	m_{32}	m_{33}	m_{34}
$3t_{11} + c_4$	m_{42}	m_{43}	m_{44}

where m_{ij} and c_i, $1 \leq i \leq 4$, $2 \leq j \leq 4$, are fixed values that depend on the passive entries and subkey values. At the end of the second round, this gives

$$\begin{aligned} C^{(2)}_{11} &= 2S(2t_{11} + c_1) + 3S(m_{22}) + S(m_{33}) + S(m_{44}) + K^{(2)}_{11} \\ &= 2S(2t_{11} + c_1) + c_5, \end{aligned}$$

for some fixed value c_5. Similarly we can get the other diagonal entries as:

$$\begin{aligned} C^{(2)}_{22} &= S(3t_{11} + c_4) + c_6 \\ C^{(2)}_{33} &= 2S(t_{11} + c_3) + c_7 \\ C^{(2)}_{44} &= S(t_{11} + c_2) + c_8 \end{aligned}$$

Since $C^{(3)}_{11} = 2C^{(2)}_{11} + 3C^{(2)}_{22} + C^{(2)}_{33} + C^{(2)}_{44} + K^{(3)}_{11}$, we can summarize the above observations with the following proposition:

Proposition 2 ([12]). *Consider a set of 256 plaintexts where the entry a_{11} is active and all the other entries are passive. Encrypt this set with 3 rounds of AES. Then, the function which maps a_{11} to $C^{(3)}_{11}$ is entirely determined by 9 fixed 1-byte parameters.*

Proof. To write the equation for $C^{(3)}_{11}$, the constants c_i, $1 \leq i \leq 8$, and $K^{(3)}_{11}$ are required. Therefore, the nine fixed values

$$\left(c_1, c_2, \ldots, c_8, K^{(3)}_{11}\right)$$

completely specify the mapping $a_{11} \rightarrow C^{(3)}_{11}$. □

Proposition 2 can be generalized: Note that the argument preceding the proposition applies to any other third round ciphertext entry and hence the statement is true for any $C^{(3)}_{ij}$. Similarly, any other a_{ij} can be taken as the active byte instead of a_{11}.

Gilbert and Minier [12] observed that the constants c_1, c_2, c_3, and c_4 depend on the values (a_{21}, a_{31}, a_{41}) on the first column, whereas the other constants c_5, c_6, c_7, and c_8 are independent of these variables. They used this information to find collisions over 3 rounds of the cipher: Assume that c_1, c_2, c_3, and c_4 behave as random functions of the variables (a_{21}, a_{31}, a_{41}). If we take about 2^{16} random (a_{21}, a_{31}, a_{41}) values and fix the other passive entries of the plaintext, by the birthday paradox, two identical functions $f, f' : a_{11} \rightarrow C^{(3)}_{11}$ will be obtained with a non-significant probability by two different values of (a_{21}, a_{31}, a_{41}). This distinguishing property was used to build attacks on AES up to 7 rounds.

Through a 1-round decryption, we get the following distinguisher for 4-round AES:

Proposition 3 ([12]). *Consider a set of 256 plaintexts where the entry a_{11} is active and all the other entries are passive. Apply 4 rounds of AES to this set. Let the function S^{-1} denote the inverse of the AES s-box and $k^{(4)}$ denote $0E \cdot K_{11}^{(4)} + 0B \cdot K_{21}^{(4)} + 0D \cdot K_{31}^{(4)} + 09 \cdot K_{41}^{(4)}$. Then,*

$$S^{-1}[0E \cdot C_{11}^{(4)} + 0B \cdot C_{21}^{(4)} + 0D \cdot C_{31}^{(4)} + 09 \cdot C_{41}^{(4)} + k^{(4)}]$$

is a function of a_{11} determined entirely by 1 key byte and 8 bytes that depend on the key and the passive entries. Thus,

$$0E \cdot C_{11}^{(4)} + 0B \cdot C_{21}^{(4)} + 0D \cdot C_{31}^{(4)} + 09 \cdot C_{41}^{(4)}$$

is a function of a_{11} determined entirely by 10 constant bytes.

4 A 5-Round Distinguisher of AES

In this section, we show how the observations of Gilbert and Minier [12] can be extended to 5 rounds. To the best of our knowledge, this is the first 5-round distinguishing property of AES. This property will help us to develop attacks on 7 rounds of AES-192 and AES-256, and on 8 rounds of AES-256.

Proposition 4. *Consider a set of 256 plaintexts where the entry a_{11} is active and all the other entries are passive. Encrypt this set with 4 rounds of AES. Then, the function which maps a_{11} to $C_{11}^{(4)}$ is entirely determined by 25 fixed 1-byte parameters.*

Proof. By Proposition 2, in the third round we have

$$\begin{aligned} C_{11}^{(3)} &= 2S(2S(2t_{11} + c_1) + c_5) + 3S(2S(2t_{11} + c_4) + c_6) \\ &\quad + S(S(t_{11} + c_3) + c_7) + S(S(t_{11} + c_2) + c_8) + K_{11}^{(3)}. \end{aligned} \tag{1}$$

Similarly it can be shown that

$$\begin{aligned} C_{22}^{(3)} &= S(S(3t_{11} + c_4) + c_9) + 2S(3S(2t_{11} + c_3) + c_{10}) \\ &\quad + 3S(S(t_{11} + c_2) + c_{11}) + S(3S(2t_{11} + c_1) + c_{12}) + K_{22}^{(3)}, \end{aligned} \tag{2}$$

$$\begin{aligned} C_{33}^{(3)} &= S(S(t_{11} + c_3) + c_{13}) + S(2S(t_{11} + c_2) + c_{14}) \\ &\quad + 2S(S(2t_{11} + c_1) + c_{15}) + 3S(2S(3t_{11} + c_4) + c_{16}) + K_{33}^{(3)} \end{aligned} \tag{3}$$

$$\begin{aligned} C_{44}^{(3)} &= 3S(S(t_{11} + c_2) + c_{17}) + S(S(2t_{11} + c_1) + c_{18}) \\ &\quad + S(3S(3t_{11} + c_4) + c_{19}) + 2S(S(t_{11} + c_3) + c_{20}) + K_{44}^{(3)}. \end{aligned} \tag{4}$$

Since

$$C_{11}^{(4)} = 2S(C_{11}^{(3)}) + 3S(C_{22}^{(3)}) + S(C_{33}^{(3)}) + S(C_{44}^{(3)}) + K_{11}^{(4)}, \tag{5}$$

the fixed values

$$\left(c_1, c_2, \ldots, c_{20}, K_{11}^{(3)}, K_{22}^{(3)}, K_{33}^{(3)}, K_{44}^{(3)}, K_{11}^{(4)}\right) \tag{6}$$

are sufficient to express the function $a_{11} \rightarrow C_{11}^{(4)}$. □

Although each of the diagonal entries depend on 9 fixed parameters, it is interesting to observe that the fourth round entry $C_{11}^{(4)}$ is entirely determined by 25 variables, rather than 37. This is a result of the fact that the constants c_1, c_2, c_3 and c_4 are common in formulas (1–4) of all the diagonal entries. Note that, like Proposition 2, Proposition 4 can also be generalized to any entry.

Since this 4-round property is related to a single entry, we can develop a 5-round distinguisher by considering the fifth round decryption:

Proposition 5. *Consider a set of 256 plaintexts where the entry a_{11} is active and all the other entries are passive. Apply 5 rounds of AES to this set. Let the function S^{-1} denote the inverse of the AES S-box and $k^{(5)}$ denote $0E \cdot K_{11}^{(5)} + 0B \cdot K_{21}^{(5)} + 0D \cdot K_{31}^{(5)} + 09 \cdot K_{41}^{(5)}$. Then,*

$$S^{-1}[0E \cdot C_{11}^{(5)} + 0B \cdot C_{21}^{(5)} + 0D \cdot C_{31}^{(5)} + 09 \cdot C_{41}^{(5)} + k^{(5)}]$$

is a function of a_{11} determined entirely by 5 key bytes and 20 bytes that depend on the key and the passive entries. Thus,

$$0E \cdot C_{11}^{(5)} + 0B \cdot C_{21}^{(5)} + 0D \cdot C_{31}^{(5)} + 09 \cdot C_{41}^{(5)}$$

is a function of a_{11} determined entirely by 26 constant bytes.

25 bytes may be too much to search exhaustively in an attack on AES-128; but for AES-256, we can precalculate and store all the possible values of this function, and using this distinguisher we can attack on 7 and 8 rounds. For AES-192, we can apply a time-memory tradeoff trick to reduce the complexity of the precomputation of the function over these 25 parameters and to make the attack feasible for 192-bit key size.

5 The Attack on AES

In this section, we describe a meet-in-the-middle attack on 7-round AES based on the distinguishing property observed in Section 4. In the attack, we first precompute all possible $a_{11} \rightarrow C_{11}^{(4)}$ mappings according to Proposition 4. Then we choose and encrypt a suitable plaintext set. We search certain key bytes, do a partial decryption on the ciphertext set, and compare the values obtained by this decryption to the values in the precomputed set. When a match is found, the key value tried is most likely the right key value. The details of the attack are as follows:

1. For each of the $2^{25\times 8}$ possible values of the parameters in (6), calculate the function $a_{11} \rightarrow C_{11}^{(4)}$, for each $0 \leq a_{11} \leq 255$, according to equations (1–4) and (5).
2. Let K_{init} denote the initial whitening subkey blocks $(K_{11}^{(0)}, K_{22}^{(0)}, K_{33}^{(0)}, K_{44}^{(0)})$. Try each possible value of K_{init}, and choose a set of 256 plaintexts accordingly to satisfy that the first entry takes every value from 0 to 255 and all other entries are fixed at the end of round 1. Also search $K_{11}^{(1)}$ to guess the value of $C_{11}^{(1)}$. Encrypt this set of plaintexts with 7 rounds of AES.
3. Let K_{final} denote the subkey blocks $(K_{11}^{(7)}, K_{24}^{(7)}, K_{33}^{(7)}, K_{42}^{(7)}, k^{(6)})$, where $k^{(6)}$ denotes $0E \cdot K_{11}^{(6)} + 0B \cdot K_{21}^{(6)} + 0D \cdot K_{31}^{(6)} + 09 \cdot K_{41}^{(6)}$. Search over all possible values of K_{final}. Using K_{final}, do a partial decryption of the ciphertext bytes $C_{11}^{(7)}$, $C_{24}^{(7)}$, $C_{33}^{(7)}$ and $C_{42}^{(7)}$ to obtain the entry $C_{11}^{(5)}$ over the set of 256 ciphertexts obtained in Step 2.
4. Now if the K_{init} and K_{final} subkeys are guessed correctly, the function $C_{11}^{(1)} \rightarrow C_{11}^{(5)}$ must match one of the functions obtained in the precomputation stage. Compare the sequence of the 256 $C_{11}^{(5)}$ values obtained in Step 3 to the sequences obtained in precomputation. If a match is found, the current key is the correct key by an overwhelming probability, since the probability of having a match for a wrong key is approximately $2^{8\times 25}\, 2^{-2048} = 2^{-1848}$.
5. Repeat the attack three times with different target values, $C_{21}^{(5)}$, $C_{31}^{(5)}$, and $C_{41}^{(5)}$, instead of $C_{11}^{(5)}$, using the same plaintext set. Having already discovered K_{init}, this attack gives us another 15 key bytes from the final two rounds.
6. Now having recovered most of the key bytes, we can search the remaining key bytes exhaustively.

This attack requires 2^{32} chosen plaintexts where the first column of the plaintext takes every possible value and the rest remain constant. There is a precomputation step which calculates 2^{200} possible values for 256 plaintexts. Therefore the complexity of this step, which will be done only once, is 2^{208} evaluations of the function. In the key search phase, for every combination of K_{init}, $K_{11}^{(1)}$, and K_{final}, we do partial decryption over 256 ciphertexts which makes 2^{88} partial decryptions. As in [7] and [10], we assume that 2^8 partial decryptions take approximately the time of a single encryption. Therefore the processing complexity of the attack is comparable to 2^{80} encryptions.

Note that since we take the target entries used in Step 5 to be on the same column as $C_{11}^{(5)}$, such as $C_{21}^{(5)}$, equations (1–4) will remain identical in these computations, and the only change will be on a few coefficients in equation (5). Hence, there won't be a need for a separate precomputation; the necessary values for $C_{11}^{(1)} \rightarrow C_{21}^{(5)}$ can be obtained with a slight overhead. However, we will need separate memory to store the obtained values. Hence, the memory requirement of the attack is $4 \times 2^{208} = 2^{210}$ bytes, which is equivalent to 2^{206} AES blocks.

5.1 A Time-Memory Tradeoff

The cost of the attack above is dominated by generation of the function set in the precomputation phase. A time-memory tradeoff approach can be useful here to balance the costs: Instead of evaluating all the possible functions in the precomputation phase, we can evaluate and store only a part of the possible function space. On the other hand, we must repeat the key search procedure a number of times with different plaintext sets to compensate the effect of this reduction. In general, if we reduce the size of the function set by a factor of n_1 and repeat the key search procedure for each candidate key n_2 times, for some $n_1, n_2 > 1$, the probability of having a match for the right key becomes, for relatively large n_1,

$$1 - \left(1 - \frac{1}{n_1}\right)^{n_2} \approx 1 - e^{-\frac{n_2}{n_1}}, \tag{7}$$

which means a success probability of 63% for $n_2 = n_1$ and 98% for $n_2 = 4n_1$.

By this tradeoff approach, one can balance different costs of the attack. The attack's complexity is currently dominated by the complexity of the precomputation phase and the required storage. As seen in Table 1, the basic attack is not feasible on AES-192. By the tradeoff approach, the precomputation cost can be reduced as desired, and the attack becomes feasible on AES-192 for $n_1 > 2^{16}$ (i.e., $n > 16$).

5.2 Extension to 8 Rounds

To attack 8 rounds of AES, we follow exactly the same steps of the 7-round attack, but we also search the last round key exhaustively. Therefore the data, precomputation, and storage complexities do not change, whereas the complexity of the key search phase increases by a factor of 2^{128}. Hence the time complexity of the attack on 8-round AES becomes 2^{208} while the memory complexity is 2^{206}. Although this attack appears to be dominated by Hellman's [13] time-memory tradeoff on both counts, it is a non-trivial attack faster than exhaustive search on 8-round AES-256.

The performance of our attacks and the previous attacks on AES are summarized in Table 1. Related key attacks, which are a different phenomenon, are not included in the comparison.

As seen in Table 1, the complexity of our attacks includes a precomputation cost in addition to the regular time complexity. The precomputation cost is considered separately from the rest of the time complexity due to the fact that it is executed only once at the time of initialization. The precomputation costs are given in terms of one evaluation of the $a_{11} \rightarrow C_{11}^{(4)}$ function according to equations (1–5).

5.3 An Improved Attack

Orhun Kara [16] observed the following improvement on the attack we described above: In the partial decryption phase of the attack in Step 3 where the attacker checks the partial ciphertext values of round 5, if the attacker looks at the XOR of two partial ciphertexts rather than looking at them individually, the $k^{(5)}$

Table 1. Plaintext, memory, time, and precomputation time complexities of the chosen plaintext attacks on AES-192 and AES-256. "MitM" stands for a meet-in-the-middle attack; "MitM-TM" denotes the time-memory tradeoff version of the attack as described in Section 5.1. Here we assume that if the precomputed set is reduced by a factor of 2^n, the key search procedure is repeated 2^{n+2} times to compensate for this reduction.

				Complexity			
Block Cipher	Paper	Rounds	Type	Data	Memory	Time	Pre.
AES-192	[12]	7	Collision	2^{32}	2^{84}	2^{140}	2^{84}
	[21]	7	Imp. Differential	2^{92}	2^{153}	2^{186}	–
	[18]	7	Square	2^{32}	2^{32}	2^{184}	–
	[10]	7	Square	$19 \cdot 2^{32}$	2^{32}	2^{155}	–
	[10]	7	Square	$2^{128} - 2^{119}$	2^{64}	2^{120}	–
	This paper	7	MitM	2^{32}	2^{206}	2^{72}	2^{208}
	This paper	7	MitM-TM	2^{34+n}	2^{206-n}	2^{74+n}	2^{208-n}
	[10]	8	Square	$2^{128} - 2^{119}$	2^{64}	2^{188}	–
AES-256	[18]	7	Square	2^{32}	2^{32}	2^{200}	–
	[12]	7	Collision	2^{32}	2^{84}	2^{140}	2^{84}
	[10]	7	Square	$21 \cdot 2^{32}$	2^{32}	2^{172}	–
	[10]	7	Square	$2^{128} - 2^{119}$	2^{64}	2^{120}	–
	[21]	7	Imp. Differential	$2^{92.5}$	2^{153}	$2^{250.5}$	–
	This paper	7	MitM	2^{32}	2^{206}	2^{72}	2^{208}
	This paper	7	MitM-TM	2^{34+n}	2^{206-n}	2^{74+n}	2^{208-n}
	[10]	8	Square	$2^{128} - 2^{119}$	2^{104}	2^{204}	–
	This paper	8	MitM	2^{32}	2^{206}	2^{200}	2^{208}
	This paper	8	MitM-TM	2^{34+n}	2^{206-n}	2^{202+n}	2^{208-n}

term in the equation cancels and he does not need to include this term in the key search.

This improved variant of the attack works as follows: In the precomputation phase, for f denoting the mapping $a_{11} \rightarrow C_{11}^{(4)}$, the attacker computes and stores,

$$S(f(i)) + S(f(0)),$$

rather than storing $f(i)$, for $1 \leq i \leq 255$. And, accordingly, in the key search phase, he looks for this XORed value in the precomputed set.

In this new variant, the key search complexity is reduced by a factor of 2^8, to 2^{72} for the 7-round attack and to 2^{200} for the 8-round attack. The complexity figures in Table 1 are given to reflect this improvement on the basic attack.

6 Conclusion

We have shown that if only one entry of a set of plaintexts is active while the other 15 entries are passive, each entry of the ciphertext after 4 rounds of AES encryption can be entirely defined using 25 fixed bytes. Using this property, we have developed the first 5-round distinguisher of AES. This enabled us to develop attacks on 7 and 8 rounds of AES-256 and 7 rounds of AES-192. The attack has a huge precomputation and memory complexity, but the data and time complexities are comparable with the best existing attacks. We have used a birthday paradox approach to reduce the precomputation and memory complexities. The proposed attacks present a new way of utilizing square-like properties for attacking AES.

Acknowledgments

We would like to thank Çağdaş Çalık for a helpful discussion on time-memory tradeoff attacks and Orhun Kara for his careful review and many valuable comments on the paper.

References

1. Biham, E., Dunkelman, O., Keller, N.: Related-key and boomerang attacks. In: Cramer, R.J.F. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 507–525. Springer, Heidelberg (2005)
2. Biham, E., Dunkelman, O., Keller, N.: Related-key impossible differential attacks on AES-192. In: Pointcheval, D. (ed.) CT-RSA 2006. LNCS, vol. 3860, pp. 21–31. Springer, Heidelberg (2006)
3. Biham, E., Keller, N.: Cryptanalysis of reduced variants of Rijndael. In: The Third AES Candidate Conference (2000)
4. Biryukov, A.: Boomerang attack on 5 and 6-round AES. In: The Fourth Conference on Advanced Encryption Standard (2004)
5. Cheon, J.H., Kim, M.J., Kim, K., Lee, J., Kang, S.: Improved impossible differential cryptanalysis of Rijndael. In: Kim, K.-c. (ed.) ICISC 2001. LNCS, vol. 2288, pp. 39–49. Springer, Heidelberg (2002)
6. Courtois, N.T., Pieprzyk, J.: Cryptanalysis of block ciphers with overdefined systems of equations. In: Zheng, Y. (ed.) ASIACRYPT 2002. LNCS, vol. 2501, pp. 267–287. Springer, Heidelberg (2002)
7. Daemen, J., Knudsen, L., Rijmen, V.: The block cipher SQUARE. In: Biham, E. (ed.) FSE 1997. LNCS, vol. 1267, pp. 149–165. Springer, Heidelberg (1997)
8. Daemen, J., Rijmen, V.: AES proposal: Rijndael. In: The First AES Candidate Conference (1998)
9. Demirci, H., Selçuk, A.A., Türe, E.: A new meet-in-the-middle attack on IDEA. In: Matsui, M., Zuccherato, R.J. (eds.) SAC 2003. LNCS, vol. 3006, pp. 117–129. Springer, Heidelberg (2004)
10. Ferguson, N., Kelsey, J., Lucks, S., Schneier, B., Stay, M., Wagner, D., Whiting, D.: Improved cryptanalysis of Rijndael. In: Schneier, B. (ed.) FSE 2000. LNCS, vol. 1978, pp. 213–230. Springer, Heidelberg (2001)
11. FIPS PUB 197. NIST
12. Gilbert, H., Minier, M.: A collision attack on 7 rounds of Rijndael. In: The Third AES Candidate Conference (2000)
13. Hellman, M.E.: A cryptanalytic time-memory trade-off. IEEE Trans. Information Theory 26(4), 401–406 (1980)

14. Hong, S., Kim, J., Lee, S., Preneel, B.: Related-key rectangle attacks on reduced versions of SHACAL-1 and AES-192. In: Gilbert, H., Handschuh, H. (eds.) FSE 2005. LNCS, vol. 3557, pp. 368–383. Springer, Heidelberg (2005)
15. Jakimoski, G., Desmedt, Y.: Related-key differential cryptanalysis of 192-bit key AES variants. In: Matsui, M., Zuccherato, R.J. (eds.) SAC 2003. LNCS, vol. 3006, pp. 208–221. Springer, Heidelberg (2004)
16. Kara, O.: Personal communication
17. Kim, J., Hong, S., Preneel, B.: Related-key rectangle attacks on reduced AES-192 and AES 256. In: Biryukov, A. (ed.) FSE 2007. LNCS, vol. 4593, pp. 225–241. Springer, Heidelberg (2007)
18. Lucks, S.: Attacking seven rounds of Rijndael under 192-bit and 256-bit keys. In: The Third AES Candidate Conference (2000)
19. Nyberg, K., Knudsen, L.R.: Provable security against a differential attack. Journal of Cryptology 8(1), 27–38 (1995)
20. Phan, R.C.W.: Classes of impossible differentials of Advanced Encryption Standard. IEE Electronics Letters 38(11), 508–510 (2002)
21. Phan, R.C.W.: Impossible differential cryptanalysis of 7-round Advanced Encryption Standard AES. Information Processing Letters 91, 33–38 (2004)
22. Phan, R.C.W., Siddiqi, M.U.: Generalized impossible differentials of Advanced Encryption Standard. IEE Electronics Letters 37(14), 896–898 (2001)
23. Zhang, W., Wun, W., Zhang, L., Feng, D.: Improved related-key impossible differential attacks on reduced round AES-192. In: Biham, E., Youssef, A.M. (eds.) SAC 2006. LNCS, vol. 4356, pp. 15–27. Springer, Heidelberg (2007)

A A Semi-square Property of AES

In this section we present a semi-square property of the AES encryption algorithm. This property observes the effect of fixing a certain bit position over the diagonal entries.

Proposition 6. *Take a set of AES plaintexts where all the non-diagonal entries are fixed. For the diagonal entries, choose a certain bit position and fix that bit of all the four diagonal entries; vary the remaining bits and obtain the set of all possible* $(2^7)^4$ *values of these plaintexts. Apply three rounds of AES to this set. Then the sum of each table entry over the ciphertext set obtained will be 0.*

One can use this semi-square property as a distinguisher to develop attacks on AES. Instead of one active entry used in the square attack, the semi-square property uses 4 semi-active entries. Therefore, the semi-square property is less efficient in terms of the required data amount. Also it is difficult to increase the number of rounds in an attack since it uses the diagonal entries. On the other hand, it is interesting to observe the effect of fixing a one-bit position. Although the s-box of AES is perfect in terms of linear and differential properties, some structural properties can still be tracked if we fix a one-bit position. This property is not preserved if we fix two or more bit positions. Understanding the mechanism behind this observation can help us to deduce more square-like properties of the cipher. This example illustrates that square properties are not restricted to just the cases where all possible values of one cell are enumerated.

Block Ciphers Implementations Provably Secure Against Second Order Side Channel Analysis

Matthieu Rivain[1,2], Emmanuelle Dottax[2], and Emmanuel Prouff[2]

[1] University of Luxembourg
[2] Oberthur Card Systems
{m.rivain,e.dottax,e.prouff}@oberthurcs.com

Abstract. In the recent years, side channel analysis has received a lot of attention, and attack techniques have been improved. Side channel analysis of second order is now successful in breaking implementations of block ciphers supposed to be effectively protected. This progress shows not only the practicability of second order attacks, but also the need for provably secure countermeasures. Surprisingly, while many studies have been dedicated to the attacks, only a few papers have been published about the dedicated countermeasures. In fact, only the method proposed by Schramm and Paar at CT-RSA 2006 enables to thwart second order side channel analysis. In this paper, we introduce two new methods which constitute a worthwhile alternative to Schramm and Paar's proposal. We prove their security in a strong security model and we exhibit a way to significantly improve their efficiency by using the particularities of the targeted architectures. Finally, we argue that the introduced methods allow us to efficiently protect a wide variety of block ciphers, including AES.

1 Introduction

Side Channel Analysis (SCA) is a cryptanalytic technique that consists in analyzing the physical leakage (called *side channel leakage*) produced during the execution of a cryptographic algorithm embedded on a physical device. SCA exploits the fact that this leakage is statistically dependent on the intermediate variables that are processed, these variables being themselves related to secret data. Different kinds of leakage can be exploited. Most of the time SCA focuses on the execution time [12], the power consumption [13] or the electromagnetic emanations [8].

Block ciphers implementations are especially vulnerable to a powerful class of SCA called *Differential* SCA (DCSA) [13,4]. Based on several leakage observations, a DSCA-attacker estimates a correlation between the leakage and different predictions on the value of a sensitive variable. According to the obtained correlation values, this attacker is able to (in)validate some hypotheses on the secret key. An alternative to DSCA exists when profiling the side channel leakage is allowed. The so-called *Profiling* SCA [6,24] is more powerful than DSCA, but it assumes a stronger adversary model. Indeed, a Profiling SCA attacker has a device under control, which he uses to evaluate the distribution of the side channel leakage according to the processed values. These estimated distributions are

K. Nyberg (Ed.): FSE 2008, LNCS 5086, pp. 127–143, 2008.

then involved in a maximum likelihood approach to recover the secret data of the attacked device. Profiling attacks are not only more efficient than DSCA but they are also more effective since they can target the key manipulations.

A very common countermeasure against SCA is to randomize sensitive variables by masking techniques [5,9]. The principle is to add one or several random value(s) (called *mask(s)*) to each sensitive variable. Masks and masked variables (together called the *shares*) propagate throughout the cipher in such a way that every intermediate variable is independent of any sensitive variable. This strategy ensures that the instantaneous leakage is independent of any sensitive variable, thus rendering SCA difficult to perform. Two kinds of masking can be distinguished: the hardware masking (that is included at the logic gate level during the design of the device) and the software masking (that is included at the algorithmic level). Hardware masking is expensive in terms of silicium area and it has some security flaws. In particular, the shares are usually processed at the same time. As a consequence the instantaneous leakage is actually dependent on the sensitive variables, which makes some dedicated attacks possible [28,19]. Other vulnerabilities come from physical phenomena such as glitches [16] or propagation delays [27]. Compared to hardware masking, software masking does not imply any overhead in silicium area, but it usually impacts the timing performances and the memory requirements. Regarding security, it does not suffer from the previous flaws and it is therefore widely used to protect block ciphers implementations.

The masking can be characterized by the number of random masks that are used per sensitive variable. A masking that involves d random masks is called a d^{th} *order masking*. When a d^{th} order masking is used, it can be broken by a $(d+1)^{th}$ *order SCA*, namely an SCA that targets $d+1$ intermediate variables at the same time. Indeed, the leakages resulting from the $d+1$ shares (*i.e.* the masked variable and the d masks) are jointly dependent on the sensitive variable. Whatever the order d, such an attack theoretically bypasses a d^{th} order masking [21]. However, the noise effects imply that the difficulty of carrying out a d^{th} order SCA in practice increases exponentially with its order [5,25] and the d^{th} *order SCA resistance* (for a given d) is thus a good security criterion for block cipher implementations.

Though block ciphers can theoretically be protected against d^{th} order SCA by using a d^{th} order masking, the actual implementation reveals some issues. The main difficulty lies in performing all the steps of the algorithm by manipulating the shares separately, while being able to re-build the expected result. As we will see, non-linear layers – crucial for the block cipher security – are particularly difficult to protect. Only a few proposals have been made regarding this issue and actually none of them provides full satisfaction. A first attempt has been made by Akkar and Goubin for the DES algorithm [2] – and improved in [1,15] – but it rests on an ad-hoc security and it is not provably secure against second order SCA. A second proposal has been made by Schramm and Paar in [25] to secure an AES implementation against d^{th} order SCA but it has been broken in [7] for $d \geq 3$. Even if it seems to be resistant for $d = 2$, its security has not been proved so that there is nowadays no countermeasure provably secure against second order SCA.

The lack of solutions implies that the higher order SCA resistance still needs to be investigated. As a first step, resistance against second order SCA (2O-SCA) is of importance since it has been substantially improved and successfully put into practice [28,19,11,18,17,14].

In this paper, we focus on block ciphers implementations provably secure against 2O-SCA. We first introduce in Sect. 2 notions about block ciphers. We recall how they are usually protected and we introduce the security model. We show that in this model, the whole cipher security can be simply reduced to the security of the S-box implementation. Then, two new generic S-box implementations are described in Sect. 3. We analyze their efficiency and we prove their security against 2O-SCA. In this section, we also propose an improvement that allows us to substantially speed up our solutions when several S-box outputs can be stored on one microprocessor word.

Because of length constraints, some results could not be included in the paper. They are given in the extended version [23]. In particular, in [23, Sect. 4] we compare our new proposal with the existing solutions, we give practical implementation results, and we discuss their requirements and their efficiency.

2 Block Ciphers Implementations Secure Against 2O-SCA

In this section, we introduce some basics about block ciphers and we explain how to implement such algorithms in order to guarantee the security against 2O-SCA. Then, we introduce a security model to prove the security of the proposed implementations.

2.1 Block Ciphers

A block cipher is a cryptographic algorithm that, from a secret key K, transforms a plaintext block P into a ciphertext block C through the repetition of key-dependent permutations, called *round transformations*. Let us denote by p, and call *cipher state*, the temporary value taken by the ciphertext during the algorithm. In practice, the cipher is *iterative*, which means that it applies R times the same round transformation ρ to the cipher state. This round transformation is parameterized by a *round key* k that is derived from K by iterating a key scheduling function α. We shall use the notations p^r and k^r when we need to precise the round r during which the variables p and k are involved: we have $k^{r+1} = \alpha(k^r, r)$ and $p^{r+1} = \rho[k^r](p^r)$, with $p^0 = P, p^R = C$ and $k^0 = \alpha(K, 0)$. Moreover, we shall denote by $(p)_j$ the j^{th} part of the state p.

The round transformation ρ can be further modeled as the composition of different operations: a key addition layer σ, a non-linear layer γ, and a linear layer λ:

$$\rho[k] = \lambda \circ \gamma \circ \sigma[k].$$

The whole cipher transformation can thus be written[1]:

$$C = \bigcirc_{r=0}^{R-1} \quad \lambda \circ \gamma \circ \sigma[k^r]\ (P).$$

Remark 1. The key scheduling function α can also be modeled as the composition of linear and non-linear layers.

The key addition layer is usually a simple bitwise addition between the round key and the cipher state and we have $\sigma[k](p) = p \oplus k$. The non-linear layer consists of several, say N, non-linear vectorial functions S_j, called *S-boxes*, that operate independently on a limited number of input bits: $\gamma(p) = \big(S_1((p)_1), \cdots, S_N((p)_N)\big)$. For efficiency reasons, S-boxes are most of the time implemented as look-up tables (LUT). We will consider in this paper that the layer λ, that mixes the outputs of the S-boxes, is linear with respect to the bitwise addition.

As an illustration, Fig. 1 represents a typical round transformation with a non-linear layer made of four S-boxes. Note that this description is not restrictive regarding the structure of recent block ciphers. In particular, this description can be straightforwardly extended to represent the AES algorithm.

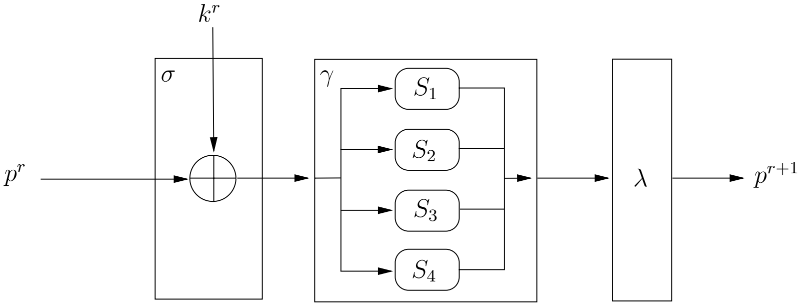


Fig. 1. A typical round transformation with a non-linear layer composed of four S-boxes

2.2 Securing Block Ciphers Against 2O-SCA

In order to obtain a 2O-SCA resistant implementation of a block cipher, we use masking techniques [5, 9]. To prevent any second order leakage, random shares are introduced: the cipher state p and the secret key k are represented by three shares – (p_0, p_1, p_2) and (k_0, k_1, k_2) respectively – that satisfy the following relations:

$$p = p_0 \oplus p_1 \oplus p_2 \ , \tag{1}$$

$$k = k_0 \oplus k_1 \oplus k_2 \ . \tag{2}$$

[1] This is not strictly the case for all iterated block ciphers. For instance, the last round of AES slightly differs from the iterated one. But this restriction does not impact on our purpose.

In order to ensure the security, shares (p_1, p_2) and (k_1, k_2) – called the masks – are randomly generated. And in order to keep track of the correct values of p and k, shares p_0 and k_0 – called the masked state and the masked key – are processed according to Relations (1) and (2).

Remark 2. For an implementation to be secure against 2O-DSCA only, the key does not need to be masked. This amounts in our description to fix the values of k_1 and k_2 at zero. In such a case, the key scheduling function can be normally implemented.

At the end of the algorithm, the expected ciphertext (which corresponds to the final value p^R) is re-built from the shares $\left(p_0^R, p_1^R, p_2^R\right)$. To preserve the security throughout the cipher and to avoid any second order leakage, the different shares must always be processed separately. Thus, the point is to perform the algorithm computation by manipulating the shares separately, while maintaining Relations (1) and (2) in such a way that the unmasked value can always be re-established. To maintain these relations along the algorithm, we must be able to maintain them throughout the three layers λ, σ and γ.

For the linear layer λ, maintaining Relations (1) and (2) is simply done by applying λ to each share separately. Indeed, by linearity, $\lambda(p)$ and the new shares $\lambda(p_i)$ satisfy the desired relation:

$$\lambda(p) = \lambda(p_0) \oplus \lambda(p_1) \oplus \lambda(p_2) \ .$$

An easy relation stands also for the key addition layer σ where each k_i can be separately added to each p_i since we have:

$$\sigma[k](p) = \sigma[k_0](p_0) \oplus \sigma[k_1](p_1) \oplus \sigma[k_2](p_2) \ .$$

For the non-linear layer, no such a relation exists and maintaining Relation (1) is a much more difficult task. This makes the secure implementation of such a layer the principal issue while trying to protect block ciphers.

Because of the non-linearity of γ, new random masks p'_1, p'_2 must be generated. Then a masked output state p'_0 has to be processed from p_0, p_1, p_2 and p'_1, p'_2 in such a way that:

$$\gamma(p) = p'_0 \oplus p'_1 \oplus p'_2.$$

Since γ is composed of several S-boxes, each operating on a subpart of p, the problem can be reduced to securely implement one S-box. The underlying problem is therefore the following.

Problem 1. Let S be an (n, m)-function (that is a function from $\mathbb{F}_2^n$ in $\mathbb{F}_2^m$). From a masked input $x \oplus r_1 \oplus r_2 \in \mathbb{F}_2^n$, the pair of masks $(r_1, r_2) \in \mathbb{F}_2^n \times \mathbb{F}_2^n$ and a pair of output masks $(s_1, s_2) \in \mathbb{F}_2^m \times \mathbb{F}_2^m$, compute $S(x) \oplus s_1 \oplus s_2$ without introducing any second order leakage.

If the problem above can be resolved by an algorithm *SecSbox*, then the masked output state p'_0 can be constructed by performing each S-box computation through this algorithm.

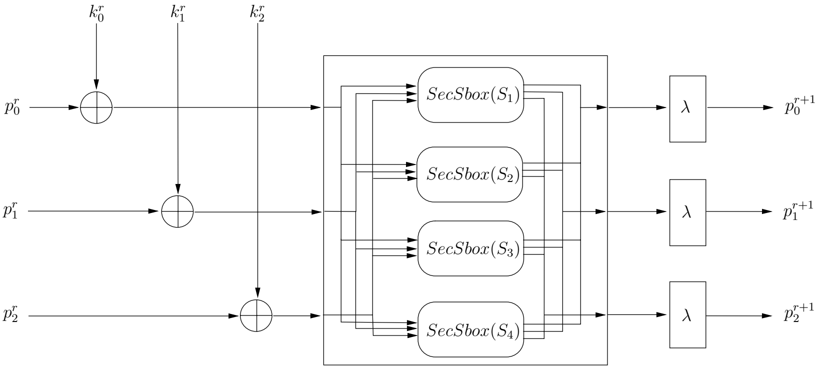


Fig. 2. A 2O-SCA resistant round transformation

Let us now assume that we have such a secure S-box implementation. Then, the scheme described in Fig. 2 can be viewed as the protected version of the round transformation described in Fig. 1. Finally, the whole block cipher algorithm protected against 2O-SCA can be implemented as depicted in Algorithm 1.

Remark 3. We have described above a way to secure a round transformation ρ. The secure implementation α_{sec} of the key scheduling function α – necessary to thwart Profiling 2O-SCA – can be straightforwardly deduced from this description since it is also composed of linear and non-linear layers.

Algorithm 1. Block Cipher secure against 2O-SCA

INPUT: a plaintext P, a masked key $k_0 = K \oplus k_1 \oplus k_2$ and the masks (k_1, k_2)
OUTPUT: the ciphertext C

```
1. $(p_1, p_2) \leftarrow rand()$
2. $p_0 \leftarrow P \oplus p_1 \oplus p_2$
3. for $r = 0$ to $R - 1$ do
4.     $(k_0, k_1, k_2) \leftarrow \alpha_{sec}((k_0, k_1, k_2), r)$
5.     $(p_0, p_1, p_2) \leftarrow (p_0 \oplus k_0, p_1 \oplus k_1, p_2 \oplus k_2)$
6.     $(p'_1, p'_2) \leftarrow rand()$
7.     for $j = 1$ to $N$ do $(p'_0)_j \leftarrow SecSbox(S_j, (p_0)_j, (p_1)_j, (p_2)_j, (p'_1)_j, (p'_2)_j)$
8.     $(p_0, p_1, p_2) \leftarrow (\lambda(p'_0), \lambda(p'_1), \lambda(p'_2))$
9. return $p_0 \oplus p_1 \oplus p_2$
```

This paper aims to design implementations that are provably secure against any kind of 2O-SCA. We will show how it can be achieved by using masking only. However, as stated in [5,26], masking must be combined with hiding-like countermeasures (such as noise addition, pipelining, operations order randomization *etc.*) to provide a satisfying resistance[2] against SCA of any order. Otherwise a higher

[2] By resistance, we mean the computational difficulty of the attack.

order SCA may be easy to carry out (see for instance [18,17]). As a consequence, to offer a good level of resistance against SCA of order greater than 2, the implementations we propose hereafter should be combined with classical hiding techniques (*e.g.* the operations order randomization described in [10] for the AES).

2.3 Security Model

In order to prove the security of our implementations, we need to introduce a few definitions. We shall say that a variable is *sensitive* if it is a function of the plaintext and the secret key (that is not constant with respect to the secret key). Additionally, we shall call *primitive random values* the intermediate variables that are generated by a random number generator (RNG) executed during the algorithm processing. In the rest of the paper, the primitive random values are assumed to be uniformly distributed and mutually independent.

In the security analysis of our proposal, we will make the distinction between a statistical dependency and what we shall call a functional dependency. Every intermediate variable of a cryptographic algorithm can be expressed as a discrete function of some sensitive variables and some primitive random values (generated by a RNG). When this function involves (*resp.* does not involve) a primitive or sensitive variable X, the intermediate variable is said to be *functionally dependent* (*resp. independent*) of X. If the distribution of an intermediate variable I varies (*resp.* does not vary) according to the value of a variable X then I is said to be *statistically dependent* (*resp. independent*) of X. It can be checked that the two notions are not equivalent since the functional independency implies the statistical independency but the converse is false. We give hereafter an example that illustrates the difference between these notions.

Example 1. Let X be a sensitive variable and let R be a primitive random value. The variable $I = X \oplus R$ is functionally dependent on X and on R. On the other hand, it is statistically independent of X since the probability $P[X = x|I = i]$ is constant for every pair (x, i).

In the rest of the paper, the term (in)dependent will be used alone to refer to the statistical notion.

Definition 1 (2O-SCA). *A* second order SCA *is an SCA that simultaneously exploits the leakages of at most 2 intermediate variables.*

From Definition 1 and according to [3,7], we formally define hereafter the security against 2O-SCA.

Definition 2. *A cryptographic algorithm is said to be* secure against 2O-SCA *if every pair of its intermediate variables is independent of any sensitive variable.*

Conversely, an algorithm is said to admit a *second order leakage* if two of its intermediate variables jointly depend on a sensitive variable.

Remark 4. Usually a second order SCA refers to an SCA that simultaneously targets two different leakage points in the time-indexed leakage vector corresponding to the algorithm execution. Thus Definitions 1 and 2 implicitly assume that an instantaneous leakage gives information on at most one intermediate variable. However, a non-careful implementation may imply that an instantaneous leakage jointly depends on two intermediate variables. This may result from physical transitions occurring at the hardware level (*e.g.* in a register or on a bus [4,20]). The different algorithms proposed in this paper fulfill security according to Definition 2 and assume a careful implementation to preclude this kind of phenomena.

Due to Definition 2, proving that an algorithm is secure against 2O-SCA can be done by listing all pairs of its intermediate variables and by showing that they are all independent of any sensitive variable. In order to simplify our security proofs, we introduce the notion of independency for a set.

Definition 3. *A set E is said to be* independent *of a variable X if every element of E is independent of X.*

By extension, Definition 3 implies that the cartesian product of two sets E_1 and E_2 is independent of a variable X if any pair in $E_1 \times E_2$ is independent[3] of X. Thus, according to Definition 2, an algorithm processing a set $\mathcal{I}$ of intermediate variables is secure against 2O-SCA if and only if $\mathcal{I} \times \mathcal{I}$ is independent of any sensitive variable.

Based on the definitions above, our security proofs make use of the two following lemmas.

Lemma 1. *Let X and Z be two independent random variables. Then, for every family $(f_i)_i$ of measurable functions the set $E = \{f_i(Z); i\}$ is independent of X.*

Remark 5. In the sequel we will say that an intermediate variable I is a function of a variable Z (namely $I = f(Z)$), if its expression involves Z and (possibly) other primitive random values of which Z is functionally independent.

Lemma 2. *Let X be a random variable defined over $\mathbb{F}_2^n$ and let R be a random variable uniformly distributed over $\mathbb{F}_2^n$ and independent of X. Let Z be a variable independent of R and functionally independent of X. Then the pair $(Z, X \oplus R)$ is independent of X.*

When a sensitive variable is masked with two primitive random values, then Lemmas 1 and 2 imply that no second order leakage occurs if the three shares are always processed separately.

According to the definitions and lemmas we have introduced, we get the following proposition.

Proposition 1. *Algorithm 1 is secure against 2O-SCA if and only if SecSbox is secure against 2O-SCA.*

Sketch of Proof. Let us denote by $\mathcal{I}$ the set of intermediate variables processed during one execution of Algorithm 1. Moreover, let us denote by $\mathcal{S}$ the set of intermediate variables processed in the different executions of *SecSbox*, and by

[3] Unlike for a set, a pair is independent of a variable X if its two elements are jointly independent of X.

$\mathcal{O}$ the set of the other intermediate variables of Algorithm 1 (namely $\mathcal{I} = \mathcal{O} \cup \mathcal{S}$). We will argue that $\mathcal{I} \times \mathcal{I}$ admits a leakage (namely a pair of $\mathcal{I} \times \mathcal{I}$ depends on a sensitive variable) if and only if $\mathcal{S} \times \mathcal{S}$ admits a leakage.

If $\mathcal{S} \times \mathcal{S}$ admits a leakage then it is straightforward that so does $\mathcal{I} \times \mathcal{I}$. Let us now show that the converse is also true. In Algorithm 1, all the operations except the S-box computations are performed independently on the three shares of every sensitive variable. This implies that $\mathcal{O} \times \mathcal{O}$ is independent of any sensitive variable *i.e.* that it admits no leakage. Since one execution of *SecSbox* takes as parameter a tuple $\big((p_0)_j, (p_1)_j, (p_2)_j, (p'_1)_j, (p'_2)_j\big)$, every intermediate variable in $\mathcal{S}$ can be expressed as a function of such a tuple. Hence, if $\mathcal{O} \times \mathcal{S}$ depends on a sensitive variable then this one is either $(p)_j$ or $(p')_j = S\big((p)_j\big)$. We deduce that the intermediate variable in $\mathcal{O}$ that jointly leaks with the one in $\mathcal{S}$ is either a share $(p_i)_j$ or a share $(p'_i)_j$. Since we have $\{(p_0)_j, (p_1)_j, (p_2)_j, (p'_0)_j, (p'_1)_j, (p'_2)_j\} \subset \mathcal{S}$ we deduce that if a leakage occurs in $\mathcal{O} \times \mathcal{S}$ then it also occurs in $\mathcal{S} \times \mathcal{S}$.

Finally, we can conclude that if a leakage occurs in $\mathcal{I} \times \mathcal{I}$ then it occurs in $\mathcal{S} \times \mathcal{S}$. ◇

In the next section, we propose two new methods to implement any S-box in a way which is provably secure against 2O-SCA. Using one of these methods as *SecSbox* in Algorithm 1 guarantees a global 2O-SCA security.

3 Generic S-Box Implementations Secure Against 2O-SCA

In this section, we first describe two methods (Sect. 3.1 and Sect. 3.2) to implement any (n, m)-function S and we prove their security against 2O-SCA. Then we propose an improvement (Sect. 3.3) that allows us to substantially reduce the complexity of both methods.

3.1 A First Proposal

The following algorithm describes a method to securely process a second order masked S-box output from a second order masked input.

Algorithm 2. Computation of a 2O-masked S-box output from a 2O-masked input

INPUT: a pair of dimensions (n, m), a masked value $\tilde{x} = x \oplus r_1 \oplus r_2 \in \mathbb{F}_2^n$, the pair of input masks $(r_1, r_2) \in \mathbb{F}_2^n \times \mathbb{F}_2^n$, a pair of output masks $(s_1, s_2) \in \mathbb{F}_2^m \times \mathbb{F}_2^m$, a LUT for the (n, m)-function S

OUTPUT: the masked S-box output $S(x) \oplus s_1 \oplus s_2 \in \mathbb{F}_2^m$

```
1. $r_3 \leftarrow rand(n)$
2. $r' \leftarrow (r_1 \oplus r_3) \oplus r_2$
3. for $a = 0$ to $2^n - 1$ do
4.     $a' \leftarrow a \oplus r'$
5.     $T[a'] \leftarrow (S(\tilde{x} \oplus a) \oplus s_1) \oplus s_2$
6. return $T[r_3]$
```

Remark 6. In the description of Step 5, we used brackets to point out that the introduction of the two output masks s_1 and s_2 is done in this very order (otherwise a second order leakage would occur).

The random value r_3 is used to mask the sum $r_1 \oplus r_2$ and to avoid any second order leakage. The value returned at the end of the algorithm satisfies: $T[r_3] = S(\tilde{x} \oplus r_3 \oplus r') \oplus s_1 \oplus s_2 = S(x) \oplus s_1 \oplus s_2$, which proves the correctness of Algorithm 2.

Complexity. Algorithm 2 requires the allocation of a table of 2^n m-bit words in RAM. It involves 4×2^n (+2) `XOR` operations, 2×2^n (+1) memory transfers and the generation of n random bits.

Security Analysis. We prove hereafter that Algorithm 2 is secure against 2O-SCA.

Security Proof. Algorithm 2 involves four primitive random values r_1, r_2, s_1 and s_2. These variables are assumed to be uniformly distributed and mutually independent together with the sensitive variable x.

The intermediate variables of Algorithm 2 are viewed as functions of the loop index a and are denoted by $I_j(a)$. The set $\{I_j(a); 0 \leq a \leq 2^n - 1\}$ is denoted by I_j. If an intermediate variable $I_j(a)$ does not functionally depend on a, then the set I_j is a singleton. The set $\mathcal{I} = I_1 \cup \cdots \cup I_{15}$ of all the intermediate variables of Algorithm 2 is listed in Table 1.

Remark 7. In Table 1, the step values refer to the lines in the algorithm description (where Step 0 refers to the input parameters manipulation). Note that one step (in the algorithm description) can involve several intermediate variables. However, these ones are separately processed and do not leak information at the same time.

In order to prove that Algorithm 2 is secure against 2O-SCA, we need to show that $\mathcal{I} \times \mathcal{I}$ is independent of x. For this purpose, we split $\mathcal{I}$ into the three subsets $E_1 = I_1 \cup \cdots \cup I_9$, $E_2 = I_{10} \cup \cdots \cup I_{14}$ and I_{15}. First, the sets $E_1 \times E_1$, $E_2 \times E_2$ and $I_{15} \times I_{15}$ are shown to be independent of x. Then, we show that $E_1 \times E_2$, $E_1 \times I_{15}$ and $E_2 \times I_{15}$ are also independent of x, thus proving the independency between $\mathcal{I} \times \mathcal{I}$ and x.

The set $E_1 \times E_1$ is independent of x since E_1 is functionally independent of x. Moreover, since $x \oplus r_1 \oplus r_2$ (*resp.* $S(x) \oplus s_1 \oplus s_2$) is independent of x and since each element in $E_2 \times E_2$ (*resp.* $I_{15} \times I_{15}$) can be expressed as a function of $x \oplus r_1 \oplus r_2$ (*resp.* $S(x) \oplus s_1 \oplus s_2$), then Lemma 1 implies that $E_2 \times E_2$ (*resp.* $I_{15} \times I_{15}$) is independent of x.

One can check that E_1 is independent of $r_1 \oplus r_2$ and is functionally independent of x. Hence, we deduce from Lemma 2 that $E_1 \times \{x \oplus r_1 \oplus r_2\}$ is independent of x, which implies (from Lemma 1) that $E_1 \times E_2$ and x are independent. Similarly, E_1 is independent of $s_1 \oplus s_2$ so that $E_1 \times \{I_{15}\}$ (namely $E_1 \times \{S(x) \oplus s_1 \oplus s_2\}$) is independent of $S(x)$ and hence of x.

Table 1. Intermediate variables of Algorithm 2

j	I_j	Steps
1	r_1	0,2
2	r_2	0,2
3	s_1	0,2
4	s_2	0,2
5	r_3	1,6
6	$r_1 \oplus r_3$	2
7	$r_1 \oplus r_2 \oplus r_3$	2,4
8	a	3,4,5
9	$a \oplus r_1 \oplus r_2 \oplus r_3$	4,5
10	$x \oplus r_1 \oplus r_2$	0,5
11	$x \oplus r_1 \oplus r_2 \oplus a$	5
12	$S(x \oplus r_1 \oplus r_2 \oplus a)$	5
13	$S(x \oplus r_1 \oplus r_2 \oplus a) \oplus s_1$	5
14	$S(x \oplus r_1 \oplus r_2 \oplus a) \oplus s_1 \oplus s_2$	5
15	$S(x) \oplus s_1 \oplus s_2$	6

To prove the independency between $E_2 \times I_{15}$ and x, we split E_2 into two subsets: $I_{10} \cup \cdots \cup I_{13}$ and I_{14}. One can check that $(x \oplus r_1 \oplus r_2, S(x) \oplus s_2)$ is independent of x and that every element of $(I_{10} \cup \cdots \cup I_{13}) \times I_{15}$ can be expressed as a function of this pair. Hence one deduces from Lemma 1 that $(I_{10} \cup \cdots \cup I_{13}) \times I_{15}$ is independent of x. In order to prove that $I_{14} \times I_{15}$ is also independent of x, let us denote $u_1 = S(x) \oplus s_1 \oplus s_2$ and $u_2 = S(x \oplus a \oplus r_1 \oplus r_2)$. The variables u_1 and u_2 are uniformly distributed[4], independent and mutually independent of x. Since $I_{14} \times I_{15}$ equals $\{S(x) \oplus u_2 \oplus u_1\} \times \{u_1\}$, we deduce that it is independent of x. ◇

3.2 A Second Proposal

In this section, we propose an alternative to Algorithm 2 for implementing an S-box securely against 2O-SCA. This second solution requires more logical operations but less RAM allocation, which can be of interest for low cost devices.

The algorithm introduced hereafter assumes the existence of a masked function *compare*$_b$ that extends the classical Boolean function (defined by *compare* $(x, y) = 0$ iff $x = y$) in the following way:

$$\mathit{compare}_b(x, y) = \begin{cases} b & \text{if } x = y \\ \bar{b} & \text{if } x \neq y \end{cases} . \tag{3}$$

Based on the function above, the second method is an adaptation of the first order secure S-box implementation which has been published in [22].

[4] This holds for u_2 if and only if the S-box S is balanced (namely every element in $\mathbb{F}_2^m$ is the image under S of 2^{n-m} elements in $\mathbb{F}_2^n$). As it is always true for cryptographic S-boxes we implicitly make this assumption.

Algorithm 3. Computation of a 2O-masked S-box output from a 2O-masked input

Input: a pair of dimensions (n, m), a masked value $\tilde{x} = x \oplus r_1 \oplus r_2 \in \mathbb{F}_2^n$, the pair of input masks $(r_1, r_2) \in \mathbb{F}_2^n \times \mathbb{F}_2^n$, a pair of output masks $(s_1, s_2) \in \mathbb{F}_2^m \times \mathbb{F}_2^m$, a LUT for the (n, m)-function S

Output: the masked S-box output $S(x) \oplus s_1 \oplus s_2 \in \mathbb{F}_2^m$

```
1. $b \leftarrow rand(1)$
2. for $a = 0$ to $2^n - 1$ do
3.     $cmp \leftarrow compare_b(r_1 \oplus a, r_2)$
4.     $R_{cmp} \leftarrow (S(\tilde{x} \oplus a) \oplus s_1) \oplus s_2$
5. return $R_b$
```

Let $indif$ denote any element in $\mathbb{F}_2^m$. Steps 3 and 4 of Algorithm 3 perform the following operations:

$$\begin{cases} cmp \leftarrow b \; ; \quad R_b \leftarrow S(x) \oplus s_1 \oplus s_2 & \text{if } a = r_1 \oplus r_2 \;, \\ cmp \leftarrow \bar{b} \; ; \quad R_{\bar{b}} \leftarrow indif & \text{otherwise.} \end{cases}$$

We thus deduce that the value returned by Algorithm 3 is $S(x) \oplus s_1 \oplus s_2$.

Complexity. The method involves 4×2^n XOR operations, 2^n memory transfers and the generation of 1 random bit. Since it also involves 2^n $compare_b$ operations, the overall complexity relies on the $compare_b$ implementation. As explained in the next paragraph, the implementation of this function must satisfy certain security properties. We propose such a secure implementation in [23, Appendix A] which – when applied to Algorithm 3 – implies a significant timing overhead compared to Algorithm 2 but requires less RAM allocation.

Security Analysis. Let δ_0 denote the Boolean function defined by $\delta_0(z) = 0$ if and only if $z = 0$. For security reasons, $compare_b(x, y)$ must be implemented in a way that prevents any first order leakage on $\delta_0(x \oplus y)$ that is, on the result of the unmasked function $compare(x, y)$ (and more generally on $x \oplus y$). Otherwise, Step 3 would provide a first order leakage on $\delta_0(r_1 \oplus r_2 \oplus a)$ and an attacker could target this leakage together with $\tilde{x} \oplus a$ (Step 4) to recover information about x. Indeed, the joint distribution of $\delta_0(r_1 \oplus r_2 \oplus a)$ and $\tilde{x} \oplus a$ depends on x which can be illustrated by the following observation: $\tilde{x} \oplus a = x$ if and only if $\delta_0(r_1 \oplus r_2 \oplus a) = 0$. In particular, the straightforward implementation $compare_b(x, y) = compare(x, y) \oplus b$ is not valid since it processes $compare(x, y)$ directly. A possible implementation of a secure function $compare_b$ is given in [23, Appendix A]. With such a function, Algorithm 3 is secure against 2O-SCA as we prove hereafter.

Security Proof. As done in Sect. 3.1, we denote by $\mathcal{I}$ the set of intermediate variables that are processed during an execution of Algorithm 3. Table 2 lists these variables. The primitive random values r_1, r_2, s_1, s_2 and b are assumed to be uniformly distributed and mutually independent together with the sensitive variable x. The following security proof is quite similar to the one done in Sect. 3.1.

Table 2. Intermediate variables of Algorithm 3

j	I_j	Steps
1	r_1	0,3
2	r_2	0,3
3	s_1	0,4
4	s_2	0,4
6	b	1,3
7	a	2-4
8	$r_1 \oplus a$	3
10	$\delta_0(a \oplus r_1 \oplus r_2) \oplus b$	3
11	$x \oplus r_1 \oplus r_2$	0,4
12	$x \oplus r_1 \oplus r_2 \oplus a$	4
13	$S(x \oplus r_1 \oplus r_2 \oplus a)$	4
14	$S(x \oplus r_1 \oplus r_2 \oplus a) \oplus s_1$	4
15	$S(x \oplus r_1 \oplus r_2 \oplus a) \oplus s_1 \oplus s_2$	4
16	$S(x) \oplus s_1 \oplus s_2$	5

In order to prove that Algorithm 3 is secure against 2O-SCA, we need to show that $\mathcal{I} \times \mathcal{I}$ is independent of x. As in Sect. 3.1 we split $\mathcal{I}$ into three subsets $E_1 = I_1 \cup \cdots \cup I_{10}$, $E_2 = I_{11} \cup \cdots \cup I_{15}$ and I_{16}. First, we show that $E_1 \times E_1$, $E_2 \times E_2$ and $I_{16} \times I_{16}$ are independent of x and then, we show that $E_1 \times E_2$, $E_1 \times I_{16}$ and $E_2 \times I_{16}$ are independent of x (thus proving that $\mathcal{I} \times \mathcal{I}$ is independent of x).

As in Sect. 3.1, $E_1 \times E_1$ is straightforwardly independent of x and the independency between $x \oplus r_1 \oplus r_2$ (*resp.* $S(x) \oplus s_1 \oplus s_2$) and x implies, by Lemma 1, that $E_2 \times E_2$ (*resp.* $I_{16} \times I_{16}$) is independent of x.

Since E_1 is independent of $r_1 \oplus r_2$ (*resp.* $s_1 \oplus s_2$) and functionally independent of x, Lemma 2 implies that $E_1 \times \{x \oplus r_1 \oplus r_2\}$ (*resp.* $E_1 \times \{S(x) \oplus s_1 \oplus s_2\}$) is independent of x. Hence, since every element of E_2 (*resp.* I_{16}) can be written as a function of $x \oplus r_1 \oplus r_2$ (*resp.* $S(x) \oplus s_1 \oplus s_2$), Lemma 1 implies that $E_1 \times E_2$ (*resp.* $E_1 \times I_{16}$) is independent of x.

Every pair in $(E_2 \backslash I_{15}) \times I_{16}$ can be expressed as a function of $(x \oplus r_1 \oplus r_2, S(x) \oplus s_2)$ which is independent of x. Hence, by Lemma 1, $(E_2 \backslash I_{15}) \times I_{16}$ is independent of x. Finally, $I_{15} \times I_{16}$ can be rewritten as $\{S(x) \oplus u_2 \oplus u_1\} \times \{u_1\}$, where u_1 $(= S(x) \oplus s_1 \oplus s_2)$ and u_2 $(= S(x \oplus r_1 \oplus r_2 \oplus a))$ are uniformly distributed, mutually independent and mutually independent of x. This implies that $I_{15} \times I_{16}$ is independent of x. ◇

3.3 Improvement

This section aims at describing an improvement of the two previous methods which can be used when the device architecture allows the storage of 2^w S-box outputs on one q-bit word (namely m, w and q satisfy $2^w m \leq q$). This situation may happen for 8-bit architectures when the S-boxes to implement have small output dimensions (*e.g.* $m = 4$ and $w = 1$) or for q-bit architectures when $q \geq 16$ (and $m \leq 8$).

In the following, we assume that the S-box is represented by a LUT having 2^{n-w} elements of bit-length $2^w m$ (instead of 2^n elements of bit-length m). This LUT, denoted by $LUT(S')$, can then be seen as the table representation of the $(n-w, 2^w m)$-function S' defined for every $y \in \mathbb{F}_2^{n-w}$ by: $S'(y) = (S(y,0), S(y,1), \cdots, S(y, 2^w - 1))$, where each $i = 0, \cdots, 2^w - 1$ must be taken as the integer representation of a w-bit value.

For every $x \in \mathbb{F}_2^n$, let us denote by $x[i]$ the i-th most significant bit of x and by x_H (*resp.* x_L) the vector $(x[1], \cdots, x[n-w])$ (*resp.* the vector $(x[n-w+1], \cdots, x[n])$). According to these notations, the S-box output $S(x)$ is the m-bit coordinate of $S'(x_H)$ whose index is the integer representation of x_L.

In order to securely compute the masked output $S(x) \oplus s_1 \oplus s_2$ from the 3-tuple $(\tilde{x}, r_1, r_2)$, our improvement consists in the two following steps. In the first step we securely compute the masked vector $S'(x_H) \oplus z_1 \oplus z_2$ (where z_1 and z_2 are $(2^w m)$-bit random masks). Then, the second step consists in securely extracting $S(x) \oplus s_1 \oplus s_2$ from $S'(x_H) \oplus z_1 \oplus z_2$.

To securely compute the masked vector $S'(x_H) \oplus z_1 \oplus z_2$, we perform Algorithm 2 (or 3) with as inputs the pair of dimensions $(n-w, 2^w m)$, the 3-tuple $(\tilde{x}_H, r_{1,H}, r_{2,H})$, the pair of output masks (z_1, z_2) and the table $LUT(S')$. This execution returns the value $S'(x_H) \oplus z_1 \oplus z_2$. Moreover, as proved in Sect. 3.1 (or Sect. 3.2), it is secure against 2O-SCA.

At this point, we need to securely extract $S(x) \oplus s_1 \oplus s_2$ from $S'(x_H) \oplus z_1 \oplus z_2$ as well as s_1 and s_2 from z_1 and z_2. Namely, we need to extract the m-bit coordinate of $S'(x_H) \oplus z_1 \oplus z_2$, and of z_1 and z_2 whose index corresponds to the integer representation of x_L. For such a purpose, we propose a process that selects the desired coordinate by dichotomy.

For every word y of even bit-length, let $H_0(y)$ and $H_1(y)$ denote the most and the least significant half part of y. At each iteration our process calls an algorithm *Select* that takes as inputs a dimension l, a 2O-masked $(2l)$-bit word $z_0 = z \oplus z_1 \oplus z_2$ (and the corresponding masking words z_1 and z_2) and a 2O-masked bit $c_0 = c \oplus c_1 \oplus c_2$ (and the corresponding masking bits c_1 and c_2). This algorithm returns a 3-tuple of l-bit words (z'_0, z'_1, z'_2) that satisfies $z'_0 \oplus z'_1 \oplus z'_2 = H_c(z)$. We detail hereafter the global process that enables to extract the 3-tuple $(S(x) \oplus s_1 \oplus s_2, s_1, s_2)$ from $(S'(x_H) \oplus z_1 \oplus z_2, z_1, z_2)$.

```
$z_0 \leftarrow S'(x_H) \oplus z_1 \oplus z_2$
**for** $i = 0$ **to** $w - 1$
    $(c_0, c_1, c_2) \leftarrow (\tilde{x}_L[w-i], r_{1,L}[w-i], r_{2,L}[w-i])$
    $(z'_0, z'_1, z'_2) \leftarrow Select(2^w m / 2^{i+1}, (z_0, z_1, z_2), (c_0, c_1, c_2))$
    $(z_0, z_1, z_2) \leftarrow (z'_0, z'_1, z'_2)$
**return** $(z_0, z_1, z_2)$
```

To be secure against 2O-SCA, this process requires that *Select* admits no second order leakage on z nor on c. A solution for such a secure algorithm is given hereafter (Algorithm 4). It requires three l-bit addressing registers (A_0, A_1), (B_0, B_1) and (C_0, C_1).

Algorithm 4.

Input: a dimension l, a masked word $z_0 = z \oplus z_1 \oplus z_2 \in \mathbb{F}_2^{2l}$, the pair of masks $(z_1, z_2) \in \mathbb{F}_2^{2l} \times \mathbb{F}_2^{2l}$, a masked bit $c_0 = c \oplus c_1 \oplus c_2 \in \mathbb{F}_2$ and the pair of masking bits $(c_1, c_2) \in \mathbb{F}_2 \times \mathbb{F}_2$

Output: a 3-tuple $(z'_0, z'_1, z'_2) \in (\mathbb{F}_2^l)^3$ that satisfies $z'_0 \oplus z'_1 \oplus z'_2 = z[c]$

```
$t_1, t_2 \leftarrow rand(l)$
$b \leftarrow rand(1)$
$c_3 \leftarrow (c_1 \oplus b) \oplus c_2$
$A_{c_3} \leftarrow H_{c_0}(z_0) \oplus t_1$
$B_{c_3} \leftarrow H_{c_0}(z_1) \oplus t_2$
$C_{c_3} \leftarrow H_{c_0}(z_2) \oplus t_1 \oplus t_2$
$A_{\overline{c_3}} \leftarrow H_{\overline{c_0}}(z_0) \oplus t_1$
$B_{\overline{c_3}} \leftarrow H_{\overline{c_0}}(z_1) \oplus t_2$
$C_{\overline{c_3}} \leftarrow H_{\overline{c_0}}(z_2) \oplus t_1 \oplus t_2$
return $(A_b, B_b, C_b)$
```

One can verify that Algorithm 4 performs the following operations for every value of (c_1, c_2):

$$\begin{cases} (A_b, B_b, C_b) \leftarrow (H_c(z_0) \oplus t_1, H_c(z_1) \oplus t_2, H_c(z_2) \oplus t_1 \oplus t_2) \\ (A_{\overline{b}}, B_{\overline{b}}, C_{\overline{b}}) \leftarrow (H_{\overline{c}}(z_0) \oplus t_1, H_{\overline{c}}(z_1) \oplus t_2, H_{\overline{c}}(z_2) \oplus t_1 \oplus t_2) \end{cases} .$$

Thus the three returned variables satisfy $A_b \oplus B_b \oplus C_b = z[c]$.

Complexity. Algorithm 4 involves 10 XOR operations and the generation of $2l + 1$ random bits.

The improvement allows to divide the execution time of Algorithm 2 (or 3) by approximately 2^w since it performs a loop of 2^{n-w} iterations instead of 2^n. Additionally, the improvement involves w calls to Algorithm 4 which implies an overhead of approximately $10 \times w$ XOR operations and the generation of $2m \times (2^w - 1) + w$ random bits. For instance, for an 8×8 S-box on a 16-bit architecture, the improvement applied to Algorithm 2 allows to save 512 XOR operations and 128 memory transfers for an overhead of 10 XOR operations and the generation of 33 random bits (16 more for (z_1, z_2) than for (s_1, s_2) and $16 + 1$ for Algorithm 4).

Security Analysis. The random values t_1 and t_2 are introduced to avoid any second order leakage on c. Otherwise, if the algorithm simply returns $(H_c(z_0), H_c(z_1), H_c(z_2))$, an inherent second order leakage (*i.e.* independent of the algorithm operations) occurs. Indeed, by targeting one of the inputs z_i and one of the outputs $H_c(z_i)$, an attacker may recover information on c since $\big(z_i, H_c(z_i)\big)$ depends on c (even if z_i is random).

The security proof of Algorithm 4 is given in the extended version of this paper [23].

4 Conclusion

In this paper, we have detailed how to implement block ciphers in a way that is provably protect against second order side channel analysis. We have introduced two new methods to protect an S-box implementation and we have proved their security in a strong and realistic security model. Furthermore, we have introduced an improvement of our methods, that can be used when several S-box outputs can be stored on one processor word. Implementation results for an 8 × 8 S-box on 16-bit and 32-bit architectures have demonstrated its practical interest [23].

Considering the today feasibility of second order attacks, our proposals constitute an interesting contribution in the field of provably secure countermeasures, as being the sole alternative to Schramm and Paar's method [25] and achieving lower memory requirements and possibly better efficiency [23].

Acknowledgements

The authors would like to thank Christophe Giraud for his valuable contribution to this work.

References

1. Akkar, M.-L., Bévan, R., Goubin, L.: Two Power Analysis Attacks against One-Mask Method. In: Roy, B., Meier, W. (eds.) FSE 2004. LNCS, vol. 3017, pp. 332–347. Springer, Heidelberg (2004)
2. Akkar, M.-L., Goubin, L.: A Generic Protection against High-Order Differential Power Analysis. In: Johansson, T. (ed.) FSE 2003. LNCS, vol. 2887, pp. 192–205. Springer, Heidelberg (2003)
3. Blömer, J., Guajardo, J., Krummel, V.: Provably Secure Masking of AES. In: Handschuh, H., Hasan, M.A. (eds.) SAC 2004. LNCS, vol. 3357, pp. 69–83. Springer, Heidelberg (2004)
4. Brier, E., Clavier, C., Olivier, F.: Correlation Power Analysis with a Leakage Model. In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 16–29. Springer, Heidelberg (2004)
5. Chari, S., Jutla, C., Rao, J., Rohatgi, P.: Towards Sound Approaches to Counteract Power-Analysis Attacks. In: Wiener, M.J. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 398–412. Springer, Heidelberg (1999)
6. Chari, S., Rao, J., Rohatgi, P.: Template Attacks. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 13–29. Springer, Heidelberg (2003)
7. Coron, J.-S., Prouff, E., Rivain, M.: Side Channel Cryptanalysis of a Higher Order Masking Scheme. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 28–44. Springer, Heidelberg (2007)
8. Gandolfi, K., Mourtel, C., Olivier, F.: Electromagnetic Analysis: Concrete Results. In: Koç, Ç.K., Naccache, D., Paar, C. (eds.) CHES 2001. LNCS, vol. 2162, pp. 251–261. Springer, Heidelberg (2001)
9. Goubin, L., Patarin, J.: DES and Differential Power Analysis – The Duplication Method. In: Koç, Ç.K., Paar, C. (eds.) CHES 1999. LNCS, vol. 1717, pp. 158–172. Springer, Heidelberg (1999)
10. Herbst, P., Oswald, E., Mangard, S.: An AES Smart Card Implementation Resistant to Power Analysis Attacks. In: Zhou, J., Yung, M., Bao, F. (eds.) ACNS 2006. LNCS, vol. 3989, pp. 239–252. Springer, Heidelberg (2006)
11. Joye, M., Paillier, P., Schoenmakers, B.: On Second-Order Differential Power Analysis. In: Rao, J.R., Sunar, B. (eds.) CHES 2005. LNCS, vol. 3659, pp. 293–308. Springer, Heidelberg (2005)

12. Kocher, P.: Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 104–113. Springer, Heidelberg (1996)
13. Kocher, P., Jaffe, J., Jun, B.: Differential Power Analysis. In: Wiener, M.J. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999)
14. Lemke-Rust, K., Paar, C.: Gaussian Mixture Models for Higher-Order Side Channel Analysis. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 14–27. Springer, Heidelberg (2007)
15. Lv, J., Han, Y.: Enhanced DES Implementation Secure Against High-Order Differential Power Analysis in Smartcards. In: Boyd, C., González Nieto, J.M. (eds.) ACISP 2005. LNCS, vol. 3574, pp. 195–206. Springer, Heidelberg (2005)
16. Mangard, S., Popp, T., Gammel, B.M.: Side-Channel Leakage of Masked CMOS Gates. In: Menezes, A. (ed.) CT-RSA 2005. LNCS, vol. 3376, pp. 351–365. Springer, Heidelberg (2005)
17. Oswald, E., Mangard, S.: Template Attacks on Masking–Resistance is Futile. In: Abe, M. (ed.) CT-RSA 2007. LNCS, vol. 4377, pp. 562–567. Springer, Heidelberg (2006)
18. Oswald, E., Mangard, S., Herbst, C., Tillich, S.: Practical Second-Order DPA Attacks for Masked Smart Card Implementations of Block Ciphers. In: Pointcheval, D. (ed.) CT-RSA 2006. LNCS, vol. 3860. Springer, Heidelberg (2006)
19. Peeters, E., Standaert, F.-X., Donckers, N., Quisquater, J.-J.: Improving Higher-Order Side-Channel Attacks with FPGA Experiments. In: Rao, J.R., Sunar, B. (eds.) CHES 2005. LNCS, vol. 3659, pp. 309–321. Springer, Heidelberg (2005)
20. Peeters, E., Standaert, F.-X., Quisquater, J.-J.: Power and Electromagnetic Analysis: Improved Model, Consequences and Comparisons. Integration 40(1), 52–60 (2007)
21. Piret, G., Standaert, F.-X.: Security Analysis of Higher-Order Boolean Masking Schemes for Block Ciphers (with Conditions of Perfect Masking). IET Information Security (to appear)
22. Prouff, E., Rivain, M.: A Generic Method for Secure SBox Implementation. In: WISA 2007. LNCS, vol. 4867, pp. 227–244 (2007)
23. Rivain, M., Dottax, E., Prouff, E.: Block Ciphers Implementations Provably Secure Against Second Order Side Channel Analysis. Cryptology ePrint Archive, Report 2008/021 (2008), http://eprint.iacr.org/
24. Schindler, W., Lemke, K., Paar, C.: A Stochastic Model for Differential Side Channel Cryptanalysis. In: Rao, J.R., Sunar, B. (eds.) CHES 2005. LNCS, vol. 3659. Springer, Heidelberg (2005)
25. Schramm, K., Paar, C.: Higher Order Masking of the AES. In: Pointcheval, D. (ed.) CT-RSA 2006. LNCS, vol. 3860, pp. 208–225. Springer, Heidelberg (2006)
26. Standaert, F.-X., Peeters, E., Archambeau, C., Quisquater, J.-J.: Towards Security Limits of Side-Channel Attacks. In: Goubin, L., Matsui, M. (eds.) CHES 2006. LNCS, vol. 4249, pp. 30–45. Springer, Heidelberg (2006)
27. Suzuki, D., Saeki, M.: Security Evaluation of DPA Countermeasures Using Dual-Rail Pre-charge Logic Style. In: Goubin, L., Matsui, M. (eds.) CHES 2006. LNCS, vol. 4249, pp. 255–269. Springer, Heidelberg (2006)
28. Waddle, J., Wagner, D.: Toward Efficient Second-order Power Analysis. In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 1–15. Springer, Heidelberg (2004)

SQUASH – A New MAC with Provable Security Properties for Highly Constrained Devices Such as RFID Tags

Adi Shamir

Computer Science department, The Weizmann Institute, Rehovot 76100, Israel
Adi.Shamir@weizmann.ac.il

Abstract. We describe a new function called *SQUASH* (which is short for *SQU*are-h*ASH*), which is ideally suited to challenge-response MAC applications in highly constrained devices such as RFID tags. It is exceptionally simple, requires no source of random bits, and can be efficiently implemented on processors with arbitrary word sizes. Unlike other ad-hoc proposals which have no security analysis, SQUASH is provably at least as secure as Rabin's public key encryption scheme in this application.

Keywords: Hash function, MAC, RFID, provable security, SQUASH.

1 Introduction

Passive RFID tags are very simple computational devices (costing a few cents each). They obtain their power from and communicate with a reader using a magnetic or electromagnetic field at a distance of several centimeters to several meters. They have many applications, including warehouse inventory control, supermarket checkout counters, public transportation passes, anti-counterfeiting tags for medicines, pet identification, secure passports, etc. They are already widely deployed, and many more applications are likely to be found in the near future.

The basic requirement in most of these applications is that a tag should be able to interactively authenticate itself securely to a reader. We assume that the tag contains some nonsecret identity I and some secret information S associated with it. When challenged by the reader, the tag sends I in the clear, and convinces the reader that it knows S, without enabling rogue eavesdroppers to extract S or to convince another reader that they know S when in fact they do not.

The classical solution for such a problem is to use zero knowledge interactive proofs, which prevent any leakage of information about S. However, such proofs are too complicated for RFID tags which have tiny memories and very limited computing power. In addition, in many applications the legitimate reader already knows the secret S, and thus we do not care about the potential leakage of information from the real prover to the real verifier. We can thus use the much simpler protocol of challenge-response authentication, in which the reader issues a random challenge R, and the tag responds with the value $H(S, R)$ where H

K. Nyberg (Ed.): FSE 2008, LNCS 5086, pp. 144–157, 2008.

is some publicly known hash function. This value, which can be viewed as a *message authentication code* (MAC), is independently computed by the reader, which accepts the authentication if and only if the computed and received values are the same.

Most of the literature on the construction of MAC's (which deals with chaining and padding techniques for multiblock inputs) is irrelevant in our challenge-response application, since we always apply H to a single block input of fixed size. The main requirement from H is that it should protect the secrecy of S even after an eavesdropper or a rogue reader gets $H(S, R_i)$ for many (known or chosen) challenges R_i. In particular, it should make it difficult for the adversary to compute the correct response of the tag to a new random challenge which had not been seen before. The function H should thus be a one-way hash function, hiding all information about S, but not necessarily a collision-resistant hash function since the discovery of a collision is not a security threat in challenge-response authentication.

Unfortunately, standard hash functions (such as SHA-1) are primarily designed to be collision resistant in order to prevent forgery of digitally signed documents. This is a very difficult requirement, which adds a lot of unnecessary complexity to their design in our application, and makes them too complicated for RFID tags. This was recognized by the RFID research community, and over the last few years there was a major effort to develop dedicated one way hash functions which are not necessarily collision resistant, and which are more suitable for RFID applications.

The best known schemes of this type belong to the HB family of schemes originally proposed by Hopper and Blum in 2001, which now includes the schemes HB [6], HB+ [7], HB++ [3], and HB-MP [11]. The security of these schemes is based on the difficulty of solving the *parity with noise problem*, which is known to be NP-complete in general, and was investigated in several recent papers [4] [8]. These schemes are much simpler than SHA-1, but they suffer from several serious problems:

1. The tag needs an internal source of random bits. Real randomness is difficult to find and can be externally manipulated by the adversary, while pseudo-randomness requires a large nonvolatile memory and lots of computations.

2. Since the proof of authenticity in these schemes is probabilistic, there is a small chance that a tag will fail to convince a reader that it is valid even when both of them are honest and there are no adversaries.

3. There are several parameters involved (the length of the secret, the number of repetitions, the probability of the additive noise, etc) and there is considerable debate about which values of these parameters will make the scheme sufficiently secure.

4. Over the last few years, a large number of attacks were developed against most of these schemes, and the various members of the HB family were developed in response to these attacks. For example, HB is known to be insecure against active adversaries. HB+ was claimed to be secure against such adversaries, but it had been recently shown in [9] that it can be attacked by a man-in-the-middle

adversary who can modify the challenges and observe the reaction of the real reader to the modified responses. With each modification, the scheme became more complicated, requiring larger keys and more computations, and it is not clear that even the latest version is completely secure.

2 The New Approach

In this paper we introduce a new function called *SQUASH* (which is a squashed form of *SQU*are-h*ASH*), which is ideally suited to RFID-based challenge-response authentication. Unlike the HB schemes it is completely deterministic, and thus it does not need any internal source of randomness and there is no way in which a legitimate tag will fail to convince a legitimate reader that it is authentic. It is exceptionally simple, and yet it is provably at least as secure as the Rabin scheme (which had been extensively studied over the last 30 years) in this application.

The basic idea of SQUASH is to mimic the operation of the Rabin encryption scheme [12], in which a message m is encrypted under key n (where the publicly known modulus n is the product of at least two unknown prime factors) by computing the ciphertext $c = m^2 \ (mod\ n)$. This is an excellent one way function, but definitely not a collision resistant function, since m, $-m$, and $m+n$ all hash to the same c. To make the Rabin scheme secure, the binary length k of n must be at least 1000 bits long, the length of m should not be much smaller than k, and thus just to store n m and c we need at least 3000 bits. Clearly we can not perform a general modular squaring operation on a severely limited RFID tag which can store less than 300 bits. Our game plan in this paper is to use the Rabin scheme as a secure starting point, and to change it in multiple ways which make it much more efficient but with provably equivalent security properties.

There were several previous attempts to simplify the implementation of modular squaring on constrained devices. At Eurocrypt 1994 [13], I proposed to replace the modular squaring operation $m^2 \ (mod\ n)$ by a randomized squaring operation $m^2 + rn$ where r is a random number which is at least 100 bits longer than n. This scheme is provably as strong as the original Rabin scheme, and has the advantage that it can be computed with a very small memory since the successive bits of m^2 and rn can be computed on the fly from LSB to MSB. This scheme can be used in low end smart cards, but it requires a lot of time and power to compute all the bits of the output (which is twice as long as in the original Rabin scheme), and is not suitable for the weaker processors contained in RFID tag.

In any challenge-response application, the secret S (which is typically 64 to 128 bits long) and the challenge R (which is typically 32 to 64 bits long) should be securely mixed and extended into a message m in the same way that keys and IV's are mixed and extended into initial states in stream ciphers. In a Rabin-based scheme, the noninvertibility of the mapping should be primarily provided by the squaring operation, and we would like to use the simplest mixing function $M(S, R)$ which addresses the known weaknesses of modular squaring, such as its easy invertibility on small inputs, its multiplicativity, and its algebraic nature (which makes it easy, for example, to compute S from $(S+R_1)^2 \ (mod\ n)$ and $(S+R_2)^2 \ (mod\ n)$ when the challenge R is numerically added to the secret S).

Studying various choices of simple mixing functions $M(S,R)$ is likely to lead to many interesting attacks and countermeasures. For example, Serge Vaudenay (in a private communication) had already developed a very clever polynomial-time attack on the case in which the short mixed value $S \oplus R$ is expanded by a *linear* feedback shift register, and then squared modulo $n = pq$.

The best choice of M also leads to a delicate theoretical dilemma: if we make it too strong (e.g., use a provably secure pseudo-random function) there is no point in squaring its result, and if we make it too weak (e.g., use a constant function) we cannot prove the formal security of the combined construction. To address this difficulty, we proceed in the rest of this paper in two different directions.

In Section 3 we assume that the choice of M is not part of the generic SQUASH construction (just as the choice of hash function for long messages is not part of the generic RSA signature scheme). We prove a *relative security result* which shows that for any choice of M, the combination $SQUASH(M(S,R))$ is at least as secure as the combination $Rabin(M(S,R))$, even though SQUASH is much simpler and faster than Rabin. More formally, we claim:

Theorem 1. *Let $\psi(S)$ be any predicate of S, which can be computed with non-negligible advantage by using a known or chosen message attack on a MAC based on the mixing function M and the SQUASH function using modulus n. Then $\psi(S)$ can be computed with at least the same advantage by the same type of attack when SQUASH is replaced by the original Rabin function with the same modulus n.*

The security of the challenge-response authentication scheme can be viewed as a special case of this theorem, in which $\psi(S)$ is defined as the value of some bit in $H(S,R)$ for a new challenge R which had not been seen before by the attacker.

In this approach, it is the responsibility of each designer to pick a particular mixing function M that he would be happy with if it would be followed by the Rabin encryption scheme, and then we give him the assurance that he would not go wrong by combining the same M with SQUASH.

Since this approach makes it difficult to evaluate the precise security and efficiency of SQUASH and to compare it to other MAC's designed for RFID applications, we propose in Section 4 a particular choice of M. Since the combined scheme has no formal proof of security, we optimize it very aggressively but we still believe that in practice it provides a high level of security at very low cost. It uses the nonlinear part of GRAIN-128[5], which is a well studied stream cipher with an extremely small footprint. Our concrete proposal (which we call SQUASH-128) is even smaller than GRAIN-128, requiring only half the number of gates to implement both M and SQUASH.

3 The Generic SQUASH Proposal

We will now describe how to simplify and speed up the Rabin encryption scheme without affecting its well studied one-wayness. The basic idea of SQUASH is to compute an excellent numerical approximation for a short window of bits in the middle of the ciphertext produced by the Rabin encryption function which uses a modulus of a particular form. We will now describe how to gradually transform Rabin to SQUASH by a series of simple observations and modifications.

Our first observation is that in the challenge-response MAC application, no one has to invert the mapping in order to recover the plaintext from the ciphertext, since both the tag and the reader compute the hash function only in the forward direction. Since we do not need a trapdoor in this application, no participant in the protocol needs to know the factorization of n, and thus everyone can use the same universal modulus n as long as no one knows how to factor it.

Our second observation is that the Rabin scheme cannot be efficiently inverted (and many of its bits can be proven secure) for *any* modulus n with unknown factorization. If a universal n with unknown factorization can be compactly represented by a small number of bits, we can save a lot of storage on the RFID tag. In particular, we recommend using a composite Mersenne number of the form $n = 2^k - 1$, which can be stored very compactly since its binary representation is just a sequence of k 1's. Other recommended choices of n which have very compact representations, such as the Cunningham project numbers of the form $n = a * (b^c) \pm d$ for small values of a, b, c, and d, will be discussed later in the paper.

A lot of effort was devoted over the last decade to determine which Mersenne numbers are prime, and to factorize those Mersenne numbers which are composite. A table summarizing the current status of these efforts is maintained by Paul Leyland [10], and the most recent success in factorizing such numbers was the complete factorization of $2^{1039} - 1$ in 2007 by a large distributed computation [1]. Since such numbers are a little easier to factor (by the special number field sieve) than general numbers (which require the general number field sieve), we recommend using numbers of the form $n = 2^k - 1$ with $1200 < k < 1300$. The currently known factors of all the "interesting" numbers in this range are summarized below. For example, $2^{1279} - 1$ is a 386 digit prime number denoted by P386, whereas $2^{1201} - 1$ has four known prime factors which are relatively small, plus a 314 decimal digit cofactor denoted by C314 which is known to be composite but has no known factors.

```
1201: 57649.1967239.8510287.2830858618432184648159211485423. C314
1213: 327511. C360
1217: 1045741327. C358
1223: 2447.31799.439191833149903. P346
1229: 36871.46703.10543179280661916121033. C339
1231: 531793.5684759.18207494497.63919078363121681207. C329
1237: C373
1249: 97423.52358081.2379005273.9345276045907272726364012481. C326
1259: 875965965904153. C365
1277: C385
1279: P386
```

```
1283: 48246753461142505411982429042143961923l9. C347
1289: 15856636079.108817410937.827446666316953.9580889333063599
.16055826953448199975207. P314
1291: 998943080897.84051400422953302189544581841. C348
1297: 12097392013313.64873964199444497. C361
```

The most interesting number in this range (and the one we recommend as the universal modulus of SQUASH) is $n = 2^{1277} - 1$, which is a 385 digit composite number with a completely unknown factorization. Another number of this type is the slightly smaller $n = 2^{1237} - 1$. Both numbers are on the "most wanted" list of computational number theorists, and a lot of effort was devoted so far to their factorization, without any success. However, there is no guarantee that these numbers will remain unfactored forever, and thus we have to consider the potential impact of either a partial or a complete factorization of the recommended modulus. As will be shown later, SQUASH is surprisingly resilient to such future events: partial factorization of $n = 2^{1277} - 1$ will have no impact on the scheme or on its formal proof of security, and even full factorization of this n will only eliminate the formal proof of security but not necessarily the real security of SQUASH. In this sense, SQUASH is much better than the original Rabin scheme, whose security will be devastated by either a partial or a full factorization of its modulus.

Our third observation is that Mersenne moduli are not only easy to store, but they also make the computation of m^2 $(mod\, n = 2^k - 1)$ particularly simple: Since $2^k = 1$ $(mod\ n)$, we just compute the double sized m^2, and then numerically add the top half to the bottom half. More precisely, if $m^2 = m1 * 2^k + m2$, then $m^2 = m1 + m2$ $(mod\ n)$. Note that this sum could be bigger than n, creating a new wraparound carry of 1, but the effect of this carry will almost certainly be limited to a few LSB bits in the result.

Our fourth observation is that there is no need to send the full 1000+ bit ciphertext c in response to the challenge R. In general, when no information about the expected response c can be computed by the adversary, the probability that the reader will accept a random t-bit answer from an adversary is 2^{-t}. In most cases, a sufficiently secure authentication of an RFID tag will be achieved if it sends $t = 32$ bits (with a cheating probability of about one in 4 billion). Low security applications can even use $t = 16$, and high security applications can either use a larger t such as 64, or repeat a low security authentication procedure several times with different challenges. The tag can thus send only a small subset of t bits from c, and as we will see shortly, sending a window of consecutive bits makes the tag's computation particularly simple. Since arithmetic modulo $2^k - 1$ has cyclic symmetry (in which rotation of the bits is equivalent to multiplication by 2), the exact location of this window within c is not important, but for the sake of concreteness in the rest of this paper we place it close to the center of c. The crucial point is that the difficulty of computing some useful predicate of the secret S (such as computing one of the bits of its expected response to some new challenge R) is *monotonically decreasing* with t since any computational task can only become easier when more information is provided in the input. In particular, if we assume that it was difficult in the original Rabin scheme then it will certainly be difficult when only t out of the k bits from each Rabin ciphertext are made available by the tag to the adversary in each response.

Our fifth observation is that if we want to be sure that a particular bit we compute in m^2 is correct, we have to compute in the worst case all the earlier bits in order to be certain about the effect of the carry entering this bit position (addition carries propagate only from LSB to MSB, so we do not have to compute higher order bits in m^2). However, we can get an excellent numeric approximation of the carry into the t bits we would actually like to compute if we compute a longer window of $t+u$ bits with u additional low order bits (which we call *guard bits*), assuming that no carry entered into the LSB of this extended window, and providing only the top t out of the $t+u$ bits as an answer. For k between 1024 and 2048, it is easy to show that the carry into each bit position in the computation of m^2 can be at most 11 bits long, and thus if we add $u = 16$ guard bits to the computed window we have only a small probability of less than $2^{11}/2^{16} = 1/32$ of computing an incorrect carry into the 17-th bit we compute. If we add $u = 64$ guard bits, then this error probability becomes negligible. Note that we can easily determine when a mistake is possible (a necessary condition is that all the top $u - 11$ guard bits above the 11 LSB bits in the extended window are 1 so that the unknown carry can propagate through them). We can thus start the computation with a small u such as 16, and only in the small fraction of the cases in which all the $u - 11$ guard bits are 1, we can rerun the computation with a larger u such as 32 or 64. This can guarantee an extremely small error probability while keeping the average running time only slightly higher than always computing $t + 16$ bits.

With this relaxation, what we gain is the ability to compute the small number of relevant bits in m^2 in linear rather than quadratic time, which is in practice one to two orders of magnitude faster than a full computation of m^2. What we lose is that the value we produce is only an approximation of the real value produced by Rabin's encryption scheme, and thus it is conceivable that by using our protocol we will reveal more information about the secret S than by using Rabin's scheme. However, the two results differ only in a negligible fraction of executions, and thus neither the reader nor the adversary is ever expected to see an incorrectly computed answer, and thus the formal security proof (based on the assumption that the Rabin scheme is secure) remains unaffected.

Our sixth observation is that if the successive bits of $m = M(S, R)$ can be efficiently generated in both the forward and backward order, we can compute the successive bits in m^2 without storing the long m explicitly, by convolving these two streams of bits. When we want to compute bit j in the lower half of m^2, we compute it by summing all the products $m_v * m_{j-v}$ for $v = 0, 1, 2, ..., j$, and add to this sum the carry from the computation of the previous bit. When we want to compute bit $j + k$ in the upper half of m^2, we compute it by summing all the products $m_v * m_{j+k-v}$ for $v = j + 1, ..., k - 1$, and add to this sum the carry from the computation of the previous bit. When we want to compute m^2

$(mod\ n)$ for $n = 2^k - 1$, we want to sum the upper half and lower half of m^2, and thus the j-th bit c_j of $c = m^2\ (mod\ n)$ can be computed by adding bits j and $j + k$ in m^2, along with their carries. It is easy to verify that the sum of the two linear convolutions defining bits j and $j + k$ is exactly the circular convolution defined as the sum of all the products $m_v * m_{j-v(mod\ k)}$ for $v = 0, 1, 2, ..., k - 1$. The final SQUASH algorithm is thus extremely simple:

1. Start with j which is the index at lower end of the desired extended window of $t + u$ bits, and set carry to 0.
2. Numerically add to the current carry (over the integers, not modulo 2) the k products of the form $m_v * m_{j-v(mod\ k)}$ for $v = 0, 1, 2, ..., k - 1$.
3. Define bit c_j as the least significant bit of the carry, set the new carry to the current carry right-shifted by one bit position, and increment j by one.
4. Repeat steps 2 and 3 $t + u$ times, throw away the first u bits, and provide the last t bits as the response to the challenge.

To implement this algorithm, we can use a simple stream cipher such as a nonlinear feedback shift register (NFSR) with a reversible state transition function, initialize it with S and R, and run it back and forth to generate all the bits of m which are used in the convolution. This requires time proportional to $k^2 * (t + u)$ which is too high for $k = 1277$. A much faster implementation uses two copies of the stream cipher in order to compute the two sequences of bits we want to circularly convolve. However, whenever the state has to wrap around (e.g., to go from the first state to the last state) it has to do so in k clock cycles. The total running time is thus proportional to $2k(t + u)$. To save an extra factor of two in the running time, we can keep one additional state in an auxiliary register. We initially load both copies of the stream cipher with the initial state, clock the second copy to the desired middle state j, and load the auxiliary register with the last state $k - 1$. We run the first copy upwards all the way from state 0 to state $k - 1$, and the second copy downwards from state j. When it reaches state 0, we exchange its state with the auxiliary register, so that now the second copy will contain state $k - 1$ and the auxiliary register will contain state 0. We continue to run the second copy downwards from state $k - 1$ to state $j + 1$. This completes the computation of the first c_j. We can now exchange the states of the first copy and the auxiliary register, and clock the second copy once, in order to bring all the components to the desired states for computing the next bit. Note that it is possible to exchange the values of two registers Y and Z without using additional storage by computing $Y = Y \oplus Z$, $Z = Y \oplus Z$, and $Y = Y \oplus Z$.

Note that due to the associativity of addition, we can compute the sum of products either upwards or downwards and get the same value, which makes it possible to run the algorithm in many different ways. For example, the first copy of the stream cipher can alternately run forwards and backwards through states $0, 1, ..., k - 1$, the second copy can alternately run backwards and forwards in a cyclic order $(mod\ k)$, incrementing its state once after each round, and the auxiliary register can alternately keep states $k - 1$ and 0 in order to help the

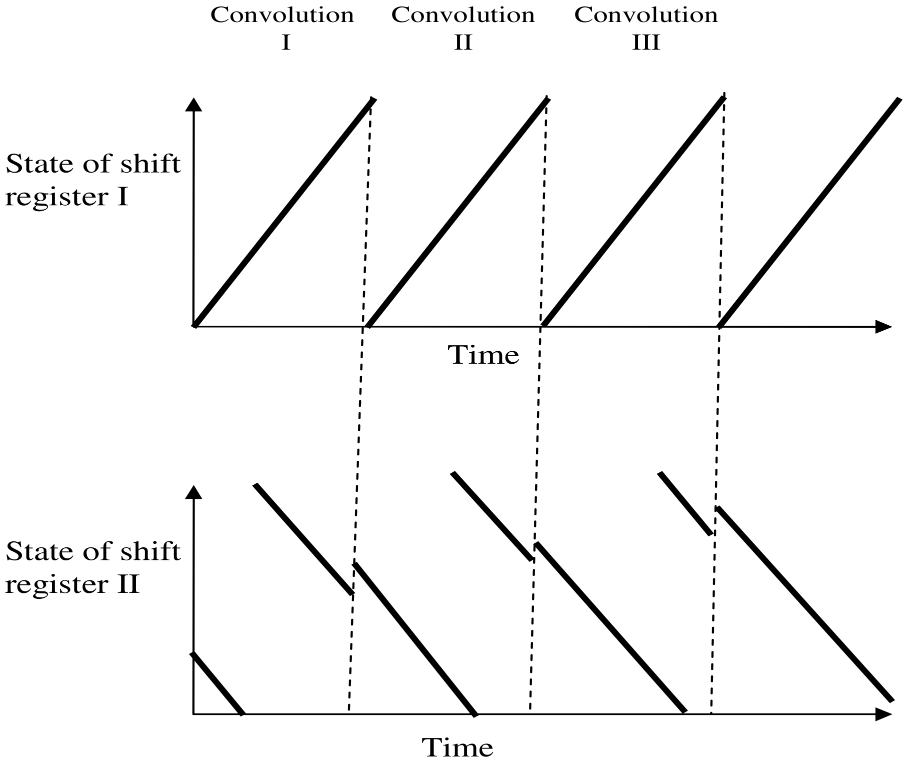


Fig. 1. The sequence of indices of the bits we have to multiply in the two streams to generate the successive output bits

second copy jump between these cyclically adjacent (but computationally wide apart) extreme states.

An important comment is that the SQUASH function is "one size fits all", and can be implemented efficiently on microprocessors with arbitrary word sizes. If the processor can multiply b-bit values in a single instruction, it can compute the same type of circular convolution b times faster by working with words rather than bits. Future RFID tags might contain simple 4-bit multipliers, which will speed up this algorithm by a factor of 4. In addition, the powerful microprocessors in the readers (which also have to carry out this computation to compare the expected and received responses) are likely to have 32-bit or even 64-bit multipliers, which will make the SQUASH algorithm extremely fast.

So far we described how to compute the SQUASH function when the underlying modulus is a composite Mersenne number of the form $2^k - 1$. It is very easy to modify the scheme to composite numbers of the form $2^k + 1$. The Rabin ciphertext in this case is defined by subtracting (instead of adding) the top half of m^2 from the bottom half. Consequently, when we compute the circular convolution we have to add to the carry all the products of the form $m_v * m_{j-v}$ for

$v = 0, 1, 2, ..., j$, and to subtract from the carry all the other products of the form $m_v * m_{j+k-v}$ for $v = j+1, ..., k-1$. Except for this minor change, everything else remains the same.

We can also consider more complicated moduli such as $n = a * 2^k - d$ where a and d are small positive integers. since $a * 2^k$ is congruent to d, we have to add to the bottom half of m^2 the top half divided by a and multiplied by d. To avoid the complicated division operation, we can change the definition of the output we are trying to compute to be a window of t consecutive bits in $a * m^2 \ (mod\ n)$. Note that the security of Rabin's encryption scheme cannot be changed if all its ciphertexts are multiplied by a known constant a, and thus we can not lose security by computing windows of bits in such modified Rabin ciphertexts instead of in the original ciphertexts. Since a multiplies both the top and the bottom parts of m^2, this implies that the algorithm now has to add to the carry all the products of the form $a * m_v * m_{j-v}$ for $v = 0, 1, 2, ..., j$, and then to add to the carry all the other products of the form $d * m_v * m_{j+k-v}$ for $v = j+1, ..., k-1$. If n is of the form $a * 2^n + d$, then the algorithm has to subtract (rather than add) from the carry all the products of the second type. When n is of the general form $a * b^c \pm d$ for small a, b, c and d, the algorithm can perform the same type of computations in base b instead of base 2, but this will probably make the scheme too complicated for a typical RFID.

Our seventh observation is that we can retain the formal proof of security even if n has some known small factors, provided that it has at least two large unknown factors. This can greatly extend the set of moduli which we can use, since most of the composite Mersenne numbers for $1000 < k < 2000$ have some small known factors. Consider, for example, the case of $n = 2^{1213} - 1$, which has a known prime factor of 327511 and a composite cofactor of 360 decimal digits whose factorization is completely unknown. If we use Rabin's encryption scheme with this n, the value of the ciphertexts modulo 327511 actually leaks the values of the plaintexts modulo this prime. We can completely stop this partial leakage of information by adding to each Rabin ciphertext a freshly selected random number between 0 and 327510, which randomizes the value of the ciphertexts modulo 327511. Since these added random values are small and we compute a window of bits near the middle of each Rabin ciphertext, we can *pretend* that such a randomizing value was indeed added to the ciphertext without changing anything in the definition of SQUASH - the only effect of such a randomization is that our numerical approximation of the middle windows in the Rabin ciphertexts will deteriorate in a negligible way[1]. An interesting corollary of this observation is that SQUASH will remain provably secure even if someone will *partially factorize* n in the future: Since we do not have to modify anything in the definition of SQUASH in order to use a modulus with a small known factor, we do not actually have to know its value when we use the scheme. Consequently, our formal proof of security will not be affected by a future discovery of some of the factors of the recommended modulus $n = 2^{1277} - 1$, provided that the

[1] This proof can be easily modified to deal with window locations which are closer to the low end of c.

factorization is partial and n has a sufficiently long cofactor whose factorization remains unknown.

Let us now assume that next year someone will find the *complete factorization* of $n = 2^{1277} - 1$. This will devastate the security of the Rabin encryption scheme which uses this modulus, since it will make it possible to decrypt all the previously produced ciphertexts. It will also eliminate the *formal proof of security* of SQUASH, but will not necessarily make it insecure in practice: Even when an attacker can extract arbitrary modular square roots *mod* n, it is not clear how to apply this operation when only a short window of bits in the middle of each Rabin ciphertext is available. In this sense, SQUASH is provably at least as secure as Rabin, but in practice it can be much more secure.

Our final observation deals with the relationship between SQUASH and some of the other proposed hash functions for RFID tags. The formal security of SQUASH is based on the difficulty of factoring the modulus n, but its implementation has the form of a cyclic convolution of a secret vector m with itself, which does not use n in an explicit way. It can thus be viewed as a scheme whose security is based on the difficulty of solving a system of quadratic equations of a very specific type. This is not entirely accurate, since the convolution is defined over the integers rather than over $GF(2)$, and the carries are defined by expressions with degrees higher than 2. In addition, the mixing function M can create complex dependencies between the bits of m. The QUAD scheme[2] is another attempt to construct a cryptographic primitive whose security is directly based on the NP-completeness of the general problem of solving systems of quadratic equations with k variables over $GF(2)$. However, the implementation complexity of QUAD is much higher than that of SQUASH since QUAD must use a dense system of quadratic equations with $O(k^2)$ randomly chosen coefficients per equation, whereas the convolution-based SQUASH has only $O(k)$ coefficients per equation defined in a very regular way. Consequently, SQUASH is much more suitable than QUAD for tiny RFID tags. Finally, HB+ also has the overall structure of convolving two vectors (S and R), but in this case R is known, and thus its security has to be based on the different problem of solving a large system of linear equations corrupted by noise.

These comparisons raise a number of interesting open problems about the security of other SQUASH-like functions. For example, SQUASH is typically implemented with two copies of the stream cipher M initialized with the same secret value and run in opposite directions. Can we initialize the two copies of M with different secret values? This can halve the number of state bits needed to get the same security against exhaustive search, but leads to bilinear rather than quadratic equations, and we have no formal argument which supports its security. Another modification is to run the two copies of M in the same direction rather than in opposite directions, and compute the dot product (with carries) of the generated sequence with various small shifts of itself. When M is implemented by a shift register, we can use only one copy of M, and get the $t + u$ shifted versions of its output by tapping various bits within this register. This can again halve the hardware complexity of the implementation, but there is no formal argument why the specific system of quadratic equations generated in such a way should be secure.

4 The Concrete SQUASH-128 Proposal

In this section we describe a fully specified MAC proposal, in order to make it possible to study its exact security and efficiency. It differs from the generic SQUASH construction in two important ways:

1. It uses a particular choice of mixing function $M_0(S, R)$, which is based on a single nonlinear feedback shift register. It shares this register with SQUASH, and thus the only additional hardware required (beyond the register itself) are a few gates to implement the feedback function and the carry adder.
2. Since the combined mapping $SQUASH(M_0(S, R))$ has no formal proof of security, we also simplify the SQUASH part in a very aggressive way by eliminating all the elements which were required by the security proof but which are not believed to contribute to the real security of the scheme. For example, we use only 8 guard bits instead of a variable number up to 64, which were needed only in order to claim that the windows of bits provided by SQUASH and Rabin are indistinguishable.

The most radical optimization step in our concrete proposal is to use a smaller modulus n. We can view the proof that SQUASH is at least as secure as Rabin as a *safety net* in order to show that the general structure of SQUASH can not be broken in polynomial time. This safety net is relatively weak (since the complexity of factoring is only subexponential in the size of n) and very erratic: it is applicable to $n = 2^{1277} - 1$ which has no known factors, but inapplicable to $n = 2^{1279} - 1$ which is a prime number. However, SQUASH seems to be much more secure than Rabin since there is no known attack on it even when the complete factorization of n is given. We believe that in fact the best attack on SQUASH requires exponential time and grows monotonically with the size of n, and thus we propose as a challenge to the reader to try to break an extremely reduced version of SQUASH which uses $n = 2^{128} - 1$ as the universal modulus, even though it is very easy to factor. We call this version SQUASH-128, and emphasize that its successful cryptanalysis will just indicate that we were too aggressive in our optimizations. The relationship between SQUASH-128 and the generic SQUASH construction is similar to the relationship between DES and the Luby-Rackoff theory of Feistel structures upon which it is loosely based.

The reduction in the size of n increases the speed of the scheme by a factor of 10, and makes it possible to halve its footprint: Since m is short, we can generate it with a single copy of M (instead of two copies which operate in opposite directions), store it in a single 128-bit register, and perform the convolutions directly on this register.

Our concrete proposal of SQUASH-128 uses a 64-bit key S and a 64-bit challenge R, and produces a 32-bit response. Our choice of M_0 is the nonlinear half of GRAIN-128 (we do not need the linear half since in this application we do not need any guaranteed lower bound on the cycle length of the generated sequence). It initializes a single 128-bit shift register denoted by $(b_0, \ldots, b_{127})$ by storing S in its low half and $S \oplus R$ in its high half. It mixes them by clocking the register 512 times (this is twice the number of initialization steps in GRAIN-128, which is still small compared to the time required by the convolutions), using no inputs and producing no outputs. The resultant 128-bit state is the value m which is squared modulo $2^{128} - 1$. The 32-bit response consists of bits 48 to 79 in the cyclically convolved result, using the 8 bits at positions 40 to 47 as guard bits. The clocking of the shift register uses the following nonlinear feedback function:

$$b_{i+128} = b_i + b_{i+26} + b_{i+56} + b_{i+91} + b_{i+96} + b_{i+3}b_{i+67} + b_{i+11}b_{i+13} + \\ b_{i+17}b_{i+18} + b_{i+27}b_{i+59} + b_{i+40}b_{i+48} + b_{i+61}b_{i+65} + b_{i+68}b_{i+84}$$

This function is the sum modulo 2 of a linear function and a quadratic bent function. It has the nice property that it can be applied up to 32 times faster by duplicating the feedback function and running these copies in parallel. Note that the zero state is a fixedpoint, and thus $S = 0$ should be excluded as a weak key.

Our choice of M_0 shares the same 128-bit shift register with SQUASH, and the only additional gates needed are an AND gate and an 8-bit carry register for the convolutions, a few AND and XOR gates for the feedback function, and two 7-bit counters for the indices v and j. Consequently, we expect the total number of gates needed by the complete SQUASH-128 scheme to be about half the number of gates in GRAIN-128, which is itself one of the smallest hardware-oriented cryptographic primitives.

This completes the description of SQUASH-128, and we encourage the reader to try to break the security of this scheme with a chosen challenge attack which requires less than 2^{64} time and space. As pointed out by Henri Gilbert and Helena Handschuh (in a private communication), this is the highest possible security level for any MAC which has a 128-bit internal state.

References

1. Aoki, K., Franke, J., Kleinjung, T., Lenstra, A.K., Osvik, D.A.: Research announcement, http://actualites.epfl.ch/presseinfo-com?id=441
2. Berbain, C., Gilbert, H., Patarin, J.: QUAD: A Practical Stream Cipher with Provable Security. In: Vaudenay, S. (ed.) EUROCRYPT 2006. LNCS, vol. 4004, pp. 109–128. Springer, Heidelberg (2006)
3. Bringer, J., Chabanne, H., Dottax, E.: HB++: a Lightweight Authentication Protocol Secure Against Some Attacks. In: Workshop on Security, Privacy and Trust in pervasive and Ubiquitous Computing - SecPerU (2006)
4. Fossorier, M.P.C., Mihaljević, M.J., Imai, H., Cui, Y., Matsuura, K.: An Algorithm for Solving the LPN Problem and Its Application to Security Evaluation of the HB Protocols for RFID Authentication. In: Barua, R., Lange, T. (eds.) INDOCRYPT 2006. LNCS, vol. 4329, pp. 48–62. Springer, Heidelberg (2006)
5. Hell, M., Johansson, T., Maximov, A., Meier, W.: A Stream Cipher Proposal: Grain-128, http://www.it.lth.se/martin/Grain128.pdf

6. Hopper, N.J., Blum, M.: A Secure Human-Computer Authentication Scheme, CMU-CS-00-139 (2000)
7. Juels, A., Weis, S.A.: Authenticating Pervasive Devices with Human Protocols. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 293–308. Springer, Heidelberg (2005)
8. Levieil, E., Fouque, P.-A.: An Improved LPN Algorithm, Security and Cryptography for Networks (2006)
9. Gilbert, H., Robshaw, M., Silbert, H.: An active attack against HB+ – a provable secure lightweight authentication protocol, Cryptology ePrint Archive number 2005/237
10. Leyland, P., http://www.leyland.vispa.com/numth/factorization/cunningham/2-.txt
11. Munilla, J., Peinado, A.: HB-MP: A further step in the HB-family of lightweight authentication protocols. Computer Networks 51, 2262–2267 (2007)
12. Rabin, M.O.: Digitalized Signatures and Public-Key Functions as Intractable as Factorization, MIT LCS/TR-212 (1979)
13. Shamir, A.: Memory Efficient Variants of Public-Key Schemes for Smart Card Applications. In: De Santis, A. (ed.) EUROCRYPT 1994. LNCS, vol. 950, pp. 445–449. Springer, Heidelberg (1995)

Differential Fault Analysis of Trivium*

Michal Hojsík[1,2] and Bohuslav Rudolf[2,3]

[1] Department of Informatics, University of Bergen, N-5020 Bergen, Norway
[2] Department of Algebra, Charles University in Prague,
Sokolovská 83, 186 75 Prague 8, Czech Republic
[3] National Security Authority, Na Popelce 2/16, 150 06 Prague 5, Czech Republic
michal.hojsik@ii.uib.no, b.rudolf@nbu.cz

Abstract. Trivium is a hardware-oriented stream cipher designed in 2005 by de Cannière and Preneel for the European project eStream, and it has successfully passed the first and the second phase of this project. Its design has a simple and elegant structure. Although Trivium has attached a lot of interest, it remains unbroken.

In this paper we present differential fault analysis of Trivium and propose two attacks on Trivium using fault injection. We suppose that an attacker can corrupt exactly one random bit of the inner state and that he can do this many times for the same inner state. This can be achieved e.g. in the CCA scenario. During experimental simulations, having inserted 43 faults at random positions, we were able to disclose the trivium inner state and afterwards the private key.

As far as we know, this is the first time differential fault analysis is applied to a stream cipher based on shift register with non-linear feedback.

Keywords: differential fault analysis, Trivium stream cipher, fault injection.

1 Introduction

In 2004 eSTREAM project has started as part of the European project ECRYPT. At the beginning there was a call for stream ciphers and 34 proposals were received. Each proposal had to be (according to the call) marked as hardware or software oriented cipher. At the time of writing this paper, the project was in phase 3, and there were just some ciphers left. One of the requirements of the call for stream ciphers was the high throughput, so the winners can compete with AES. In this respect, one of the best proposals is the stream cipher Trivium, which is a hardware oriented stream cipher based on 3 nonlinear shift registers. Though the cipher has a hardware oriented design it is also very fast in software, which makes it one of the most attractive candidates of the eSTREAM project.

In this paper differential fault analysis of Trivium is described. We suppose that an attacker can corrupt a random bit of the inner state of Trivium.

* The work was partly supported by the grant from GAUK n. 7302/2007.

K. Nyberg (Ed.): FSE 2008, LNCS 5086, pp. 158–172, 2008.

Consequently some bits of keystream difference (proper keystream XOR faulty keystream) depend linearly on the inner state bits, while other equations given by the keystream difference are quadratic or of higher order in the bits of the fixed inner state.

Since we suppose that an attacker can inject a fault only to a random position, we also describe a simple method for fault position determination. Afterwards knowing the corresponding faulty keystream, we can directly recover few inner state bits and obtain several linear equations in inner state bits. Just by repeating this procedure for the same inner state but for different (randomly chosen) fault positions we can recover the whole cipher inner state, and clocking it backwards we are able to determine the secret key. The drawback of this simple approach is that we need many fault injections to be done in order to have enough equations.

To decrease number of faulty keystreams needed (i.e. to decrease the number of fault injections needed), we also use quadratic equations given by a keystream difference. But as we will see later on, we do not use all quadratic equations, but just those which contains only quadratic monomials of a special type, where the type follows directly from the cipher description. In this way we are able to recover the whole trivium inner state using approx. 43 fault injections.

As mentioned above, presented attacks require many fault injections to the same Trivium inner state. This can be achieved in the chosen-ciphertext scenario, assuming that the initialisation vector is the part of the cipher input. In this case, an attacker will always use the same cipher input (initialisation vector and ciphertext) and perform the fault injection during the deciphering process. Hence, proposed attacks could be described as chosen-ciphertext fault injection attacks.

We have to stress out, that in this paper we do not consider usage of any sophisticated methods for solving systems of polynomial equations (e.g. Gröbner basis algorithms). We work with simple techniques which naturally raised from the analysis of the keystream difference equations. Hence the described attacks are easy to implement. This also shows how simple is to attack Trivium by differential fault injection. We believe that usage of more sophisticated methods can further improve presented attack, in sense of the number of the fault injections needed for the key recovery.

The rest of this paper is organised as follows. In Sect. 2 we review related work and Sect. 3 describes used notation. Trivium description in Sect. 4 follows. Attacks description can be found in Sect. 5, which also contains attack outline and differential fault analysis description. We conclude by Sect. 6.

2 Related Work

Let us briefly mention some of the previous results in Trivium cryptanalysis. Raddum introduces a new method of solving systems of sparse quadratic equations and applies it in [2] to Trivium. The complexity arising from this attack on Trivium is O(2^{162}). A. Maximov and A. Biryukov [3] use a different approach to solve the system of equations produced by Trivium by guessing the value of some bits. In some cases this reduces the system of quadratic equations to a system of linear equations that can be solved. The complexity of this attack is $O(c \cdot 2^{83,5})$, where c is the time taken to solve a sparse system of linear equations. Different approaches to Trivium potential cryptanalysis - some ways of construction and solution of equations system for Trivium mainly - are discussed in [4]. M. S. Thuran and O. Kara model the initialisation part of Trivium as an 8-round function in [5]. They study linear cryptanalysis of this part of Trivium and give a linear approximation of 2-round Trivium with bias 2^{-31}. In [6] the differential cryptanalysis is applied mainly to initialisation part of Trivium.

Our attack deals with fault analysis of Trivium. Side-channel attacks are amongst the strongest types of implementation attacks. Short overview on passive attacks on stream ciphers is given in [7]. Differential power analysis of Trivium is described in [8].

Fault attacks on stream ciphers are studied in [9]. The authors are mainly focused on attacking constructions of stream ciphers based on LFSRs. The corresponding attacks are based on the linearity of the LFSR. More specialised techniques were used against specific stream ciphers such as RC4, LILI-128 and SOBER-t32 [10].

3 Notation

In this paper the inner state of Trivium is denoted as IS and the bits of the inner state (there are 288 of these) as $(s_1, \ldots, s_{288})$. The inner state at time t_0 is denoted as IS_{t_0} and the following keystream (starting at the time t_0) as $\{z_i\}_{i=1}^{\infty}$. We refer to this keystream as the proper keystream. After a fault injection into the state IS_{t_0}, the resulting inner state is denoted as IS'_{t_0} and the following faulty keystream (starting at the time t_0) as $\{z'_i\}_{i=1}^{\infty}$.

The keystream difference, i.e. the difference between the proper keystream and the faulty keystream is denoted as $\{d_i\}$, i.e. $d_i = z'_i \oplus z_i,\ i \geq 1$. The fault position is denoted as e, $1 \leq e \leq 288$. The righ-hand-side of an equation is (as usual) abbreviated to RHS.

4 Trivium Description

The stream cipher Trivium is an additive synchronous stream cipher with 80-bit secret key and 80-bit initialisation vector (IV). The cipher itself produces the keystream, which is then XOR-ed to a plaintext to produce the ciphertext. Trivium (as other stream ciphers) can be divided into two parts: the *initialisation algorithm* described by Alg. 1, which turns the secret key and the initialisation vector into the inner state of Trivium, and the *Keystream generation algorithm* described by Alg. 2, which produces the keystream (one bit per step).

The cipher itself consists of 3 shift registers with non-linear feedback functions. These registers are of length 93, 84 and 111 respectively. Keystream production function is a bit sum (i.e. XOR) of 6 bits in total, 2 bits from each register. Feedback function for register i $(i = 0, 1, 2)$ depends on bits of register i quadratically and on one bit of register $(i+1) \bmod 3$ linearly. If we look closer at any of these feedback functions we see, that it contains only one quadratic term and the rest is linear. Furthermore, this quadratic term is of a special type, namely $s_j \cdot s_{j+1}$.

In the rest of the paper, by the term *pair quadratic equation* we denote a quadratic equation, which contains linear terms and quadratic terms only of the type $s_j \cdot s_{j+1}$. As we will see, these pair quadratic equations are typical for Trivium and in our attack we take an advantage of this.

Algorithm 1. The Initialisation Algorithm of Trivium

Input: Secret key $K = (K_1, \ldots, K_{80})$, initialisation vector $IV = (IV_1, \ldots, IV_{80})$
Output: Trivium inner state $(s_1, \ldots, s_{288})$

```
$(s_1, \ldots, s_{93}) \leftarrow (K_1, \ldots, K_{80}, 0, \ldots, 0)$
$(s_{94}, \ldots, s_{177}) \leftarrow (IV_1, \ldots, IV_{80}, 0, \ldots, 0)$
$(s_{178}, \ldots, s_{288}) \leftarrow (0, \ldots, 0, 1, 1, 1)$
for $i = 0$ to $4 \cdot 288$ do
    $t_1 \leftarrow s_{66} + s_{91} \cdot s_{92} + s_{93} + s_{171}$
    $t_2 \leftarrow s_{162} + s_{175} \cdot s_{176} + s_{177} + s_{264}$
    $t_3 \leftarrow s_{243} + s_{286} \cdot s_{287} + s_{288} + s_{69}$
    $(s_1, \ldots, s_{93}) \leftarrow (t_3, s_1, \ldots, s_{92})$
    $(s_{94}, \ldots, s_{177}) \leftarrow (t_1, s_{94}, \ldots, s_{176})$
    $(s_{178}, \ldots, s_{288}) \leftarrow (t_2, s_{178}, \ldots, s_{287})$
end for
```

Algorithm 2. The Keystream generation algorithm

Input: Trivium inner state $(s_1, \ldots, s_{288})$, number of output bits $N \leq 2^{64}$
Output: Keystream $\{z_i\}_{i=1}^{N}$

```
for $i = 0$ to $N$ do
    $z_i \leftarrow s_{66} + s_{93} + s_{162} + s_{177} + s_{243} + s_{288}$
    $t_1 \leftarrow s_{66} + s_{91} \cdot s_{92} + s_{93} + s_{171}$
    $t_2 \leftarrow s_{162} + s_{175} \cdot s_{176} + s_{177} + s_{264}$
    $t_3 \leftarrow s_{243} + s_{286} \cdot s_{287} + s_{288} + s_{69}$
    $(s_1, \ldots, s_{93}) \leftarrow (t_3, s_1, \ldots, s_{92})$
    $(s_{94}, \ldots, s_{177}) \leftarrow (t_1, s_{94}, \ldots, s_{176})$
    $(s_{178}, \ldots, s_{288}) \leftarrow (t_2, s_{178}, \ldots, s_{287})$
end for
```

5 Differential Fault Analysis of Trivium

In this section we will describe our contribution to the cryptanalysis of Trivium. To show how our attack evolved, we describe more versions of differential fault analysis of Trivium, from the simplest one, to the more sophisticated ones.

Before describing our contribution, let us briefly recall the basic ideas of Differential Fault Analysis (DFA).

Differential fault analysis is an active side channel attack technique, in which an attacker is able to insert a fault into the enciphering or deciphering process or he is able to insert a fault into a cipher inner state. The later is the case of our attack, we suppose that an attacker is able to change exactly one bit of the Trivium inner state. Another assumption of DFA is that an attacker is able to obtain not only the cipher output after the fault injection, but he is also able to obtain the standard output, i.e. the output produced by the cipher without the fault injection.

In this paper, according to the DFA description, we assume that an attacker is able to obtain a part of a Trivium keystream $\{z_i\}_{i=1}^{\infty}$ produced from the arbitrary but fixed inner state IS_{t_0} and that he is also able to obtain a part of the faulty keystream $\{z_i'\}_{i=1}^{\infty}$ produced by the fault inner state IS_{t_0}'. We will discuss the amount of keystream bits needed for any of presented attacks later on. Just for illustration, the attack we have implemented needs about 280 bits of the proper keystream and the same amount of the faulty keystream bits.

Furthermore, in our attacks, an attacker has to be able to do the fault injection many times, but for the same inner state IS_{t_0}, where the value of t_0 is fixed, but arbitrary and unknown. It follows, that in our scenario the stream cipher Trivium has to be run many times with the same key and IV, so an attacker is able to inject a fault to the same inner state IS_{t_0}. This can be achieved e.g. by attacking the cipher in the deciphering mode, assuming the initialisation vector is the part of the cipher input. In this case, we will always use the same pair of IV and cipher text as the cipher input, and we will perform fault injections to the cipher inner state during the deciphering process. Hence, proposed attacks can be performed in the chosen-ciphertext attack scenario.

All our prerequisites are gathered in Sect. 5.1.

The result of our attack is the determination of the inner state IS_{t_0}, which can be used afterwards to obtain the secret key K and initialisation vector IV. This can be done due to the reversibility of the Trivium initialisation algorithm and due to the fact, that the initialisation part is the same as the keystream generation part. Thus, after we determine IS_{t_0}, we run trivium backwards (which also allows us to decipher previous communication) until we obtain an inner state of the form

$$\begin{aligned}(s_1, \ldots, s_{93}) &= (a_1, \ldots, a_{80}, 0, \ldots, 0),\\ (s_{94}, \ldots, s_{177}) &= (b_1, \ldots, b_{80}, 0, \ldots, 0),\\ (s_{178}, \ldots, s_{288}) &= (0, \ldots, 0, 1, 1, 1)\end{aligned} \tag{1}$$

Afterwards if the values $(b_1, \ldots, b_{80})$ are equal to the known IV, we can (with very high probability) claim, that $(a_1, \ldots, a_{80})$ is the secret key used for encryption.

5.1 Attack Prerequisites

Let t_0 be arbitrary but fixed positive integer and let IS_{t_0} be arbitrary but fixed Trivium inner state at time t_0. These are the prerequisites of our attack:

- an attacker is able to obtain first n consecutive bits of the keystream $\{z_i\}$ produced out of the inner state IS_{t_0},
- an attacker is able to inject exactly one fault at random position of the inner state IS_{t_0},
- an attacker is able to repeat the fault injection at random position of IS_{t_0} m times
- an attacker is able to obtain first n consecutive bits of the keystream $\{z'_i\}$ produced out of the inner sate IS'_{t_0} for all fault injections.

The number of consecutive bits of the proper and of the faulty keystream needed, n, differs for presented attacks. For the most simple one $n = 160$ and for the second one $n = 280$. The number of fault injections needed for any of the presented attacks, m, is discussed after the attack descriptions.

5.2 Attack Outline

The core of the presented attack is to solve the system of equations in the inner state bits of a fixed inner state $IS_{t_0} = (s_1, \ldots, s_{288})$. Because the output function of the Trivium is linear in the inner state bits, some equations can be obtained directly from the proper keystream. Specifically, the first 66 keystream bits are linear combinations of bits of IS_{t_0},

$$z_i = s_{67-i} + s_{94-i} + s_{163-i} + s_{178-i} + s_{244-i} + s_{289-i},\ 1 \leq i \leq 66, \qquad (2)$$

and the following 82 keystream bits depends quadratically on $s_1, \ldots, s_{288}$. These are followed by polynomials of degree 3 and higher.

Since we have 288 variables (bits of inner state) and the degree of keystream equations grows very fast, we are not able to efficiently solve this polynomial system. The question is how to obtain more equations. By the analysis of Trivium, we have noticed that a change of a single bit of the inner state directly reveals some inner state bits and also gives us some more equations. Hence, we decided to use DFA as a method for obtaining more equations. For illustration, Tab. 1 contains the first equations given by a keystream difference after a fault injection on position 3, i.e. $s'_3 = s_3 + 1$.

Table 1. Non-zero elements of $\{d_i\}_{i=1}^{230}$ for $s'_3 = s_3 + 1$

i	d_i
64,91,148,175,199,211,214,217	1
158,175,224	s_4
159,174,225	s_3
212	$s_4 + s_{31} + s_{29}s_{30} + s_{109}$
213	$s_2 + s_{29} + s_{27}s_{28} + s_{107}$
227	$s_{178} + s_{223} + s_{221}s_{222} + s_4$
228	$s_{161} + s_{176} + s_{174}s_{175} + s_{263} + s_{221} + s_2 + s_{219}s_{220}$

5.3 Fault Position Determination

In our attack we suppose that an attacker is not able to influence the position of a fault injection, i.e. he can inject a fault to the Trivium inner state only at a random position. But as we will see further on, he also needs to determine the fault position, since the equations for the keystream difference depend on this position.

The core of the fault position determination is that the distance between the output bits differs for each register. According to the line 2 of Alg. 2, s_{66} and s_{93} are the output bits of the first register and their distance is $93 - 66 = 27$. In the second register, s_{162} and s_{177} are used as the output bits and their distance is 15. In the third register s_{243} and s_{288} are used and their distance equals 45.

For example suppose that we have injected a fault into one of the registers at an unknown position e, so $s'_e = s_e + 1$, and (only for this example) assume that we know that $e \in \{1, \ldots, 66\} \cup \{94, \ldots, 162\} \cup \{178, \ldots, 243\}$. Denote the index of the first non-zero bit in the keystream difference by a, i.e. $d_a = 1$ and $d_j = 0$ for all $1 \leq j < a$. If the fault was injected into the first register, then according to the output function we have also $d_{a+27} = 1$, since the distance between the output bits of the first register is 27. In the same manner, if the fault was injected to the second register, we have $d_{a+15} = 1$ while $d_{a+27} = 0$, and $d_{a+45} = 1$ while $d_{a+15} = d_{a+27} = 0$ for the third register.

A non-zero bit can occur in the keystream difference at many positions, depending on the values of inner state bits. But since some occurrences of the non-zero bit are certain, with a little bit more work that in our example, we can easily determine the exact fault position. Tables 6, 7 and 8 in Appendix show the positions and the values of the some first (potentially) nonzero bits of the keystream difference in the correspondence to the fault position. In these tables, symbol X denotes a value which is neither 1 nor s_{i+1} or s_{i-1}. By the help of these tables, it is easy to determine the fault position assuming that exactly one fault was injected (our assumption from 5.1). E.g. assume that the first non-zero keystream difference bit has index a, the second non-zero bit has index b and that $b - a = 42$. According to the tables 6, 7 and 8, we see that this can happen only in the case described by the third row of Tab. 6 $(136 - 94 = 42)$ and in the case described by the third row of Tab. 8 $(331 - 289 = 42)$. In the first case we will also have $d_{b+42} = 1$ $(178 - 136 = 42)$ and $d_{b+24} = 0$, while in the second case $d_{b+42} = 0$ and $d_{b+24} = 1$ $(355 - 331 = 24)$. In this way we can distinguish between the two cases and we can claim that the fault position is $94 - a$ in the first case and $289 - a$ in the second case.

5.4 First Attack, Using Linear Equations

Let us start with the description of a simple attack on Trivium using fault injection technique.

In this attack an attacker uses only linear equations in the inner state bits $(s_1, \ldots, s_{288})$ given by the proper keystream and by the keystream difference.

Before the attack itself, the attacker does the following precomputation: for each fault position $1 \leq e \leq 288$, the attacker expresses potentially non-zero bits

Table 2. The average number (among all fault positions) of equations obtained from a random fault

number of steps	The average number of equations of degree d obtained from one fault.						
	$d = 1$	$d = 2$	$d = 3$	$d = 4$	$d = 5$	$d = 6$	$d = 7$
160	1.86	0.24	0.08	0	0	0	0
180	1.99	1.17	0.39	0	0	0	0
200	1.99	2.52	0.89	0	0	0	0
220	1.99	4.14	1.53	0	0	0	0
240	1.99	5.99	2.82	0.03	0	0	0
260	1.99	7.76	4.15	1.13	0.45	0.37	0.28
280	1.99	9.22	5.22	3.42	1.47	1.23	0.96
300	1.99	9.77	5.86	7.10	3.55	2.66	2.09

of the keystream difference as polynomials in $(s_1, \ldots, s_{288})$ over GF(2), using Alg. 2 and the fact that $d_i = z'_i \oplus z_i$. Since he only needs linear equations, the attacker has to express only the first n bits of keystream difference and store these equations in a table. The value of n is discussed below. During the attack, the attacker will just make table look-up for the right equation for the actual fault position.

The average number of linear equations given by the keystream difference for a single fault injection can be found in Tab. 2. It follows from this table, that in the precomputation phase of this attack, it is enough to make 180 steps of Trivium (in a symbolic computation) for each fault position, so $n = 180$.

Let us have a closer look on the relation between the number of fault injections and the number of inner state bits obtained. At the beginning of the attack, almost every fault injected gives us directly two new variables. But as the attack progresses, it becomes much harder to hit the positions which will bring us two new inner state bits and there will be many fault injections, which bring only one or even no new variable. If we would like to obtain all of the 288 bits of the inner state just by the fault injection, at the end of the attack we will waste many fault injections until we hit the right positions. Hence, it will significantly reduce the number of fault injections needed, if we stop the attack at the point when T bits of inner state are known and we will guess the remaining $288 - T$ bits afterwards.

Table 3 shows the number of fault injections needed, m, to obtain T bits of inner state for different values of T. This is also illustrated on the left of Fig. 1.

During the attack, the attacker stores linear equations obtained in a binary matrix M with 289 columns (288 bits of IS_{t_0} plus RHS). The attack itself is described by Alg. 3.

Table 3. Number of fault injections needed (m) to obtain a certain number of inner state bits (T) (average over 1000 experiments) in the linear attack

T	20	40	60	80	100	120	140	160	180	200	220	240	260	280	288
m	10	21	33	46	58	71	84	98	113	127	145	165	189	270	382

Algorithm 3. Linear attack

```
Input: Trivium stream cipher with possibility of fault injections (see Sect. 5.1)
Output: Secret key $K$
obtain $n$ consecutive bits of $\{z_i\}$
insert linear equations (2) (see Sect. 5.2) to $M$, using bits of $\{z_i\}$ as RHS
while rank$(M) < T$ do
    insert a fault into $IS_{t_0}$
    obtain $n$ consecutive bits of the faulty keystream $\{z'_i\}$
    compute keystream difference $d_i = z'_i + z_i,\ 1 \leq i \leq n$
    determine the fault position $e$
    insert equations for the keystream difference (according to value of $e$) into $M$
    do Gauss elimination of $M$
end while
repeat
    guess the remaining $288 - T$ inner state bits
    solve the linear system given by $M$ and guessed bits
    store the solution in $IS_S$
    produce the keystream $\{\tilde{z}_i\}_{i=1}^{n}$ from the inner state $IS_S$
until $\exists\ i \in \{1, \ldots, n\} : \tilde{z}_i \neq z_i$
run Trivium backwards starting with $IS_S$ until an inner state $IS_0 = (s_1^0, \ldots, s_{288}^0)$
    of type (1) (see page 162) is reached
output $K = (s_1^0, \ldots, s_{80}^0)$.
```

As already mentioned, according to Tab. 2, in Alg. 3 we can set $n = 180$, since we would not get any more (previously unknown) linear equations from the bits of the keystream difference $\{d_i\}$ for $i > 180$.

The complexity of this simple attack is given by the complexity of solving a system of linear equations (suppose this is $\mathrm{O}(n^3)$) multiplied by the complexity of guessing $288 - T$ variables. For $T = 258$, we obtain the complexity of $288^3 \cdot 2^{30} = 2^{54}$ operations and we need to do (according to Tab. 3) approximately 189 fault injections. For $T = 268$, the attack has the complexity of $2^{44.2}$ operations and we need to do approximately 200 fault injections.

5.5 Second Attack, Using Pair Quadratic Equations

The attack presented in this section is a natural successor of the previous one, in the terms of reduction of the number of fault injections needed.

The main difference is that in this case, we do not use only linear equations but also *pair* quadratic equations (see Sect. 4). The reason why we have decided to use only pair quadratic and not all quadratic equations is that most of quadratic equations that appear in Trivium analysis are pair equations. For example all 82 quadratic equations for keystream bits are pair and also most (approx. 80%) of the quadratic equations for the keystream difference bits are also pair (see Tab. 4). Furthermore, the number of all possible quadratic terms in 288 inner state bits is too large (approx. $2^{16.3}$) and hence the complexity of solving a linear system in all quadratic variables would be too high.

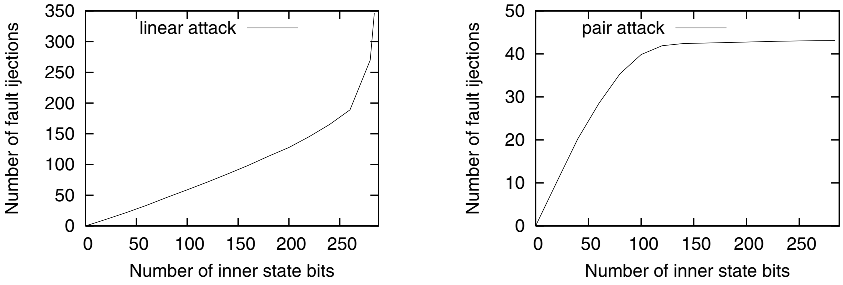


Fig. 1. Number of fault injections needed to obtain a certain number of inner state bits. Left: linear attack. Right: pair quadratic attack.

Lets have a look at the precomputation part of this attack. Here again, for all possible fault positions $1 \leq e \leq 288$, we need to express bits of the keystream difference as polynomials in the bits of $IS_{t_0} = (s_1, \ldots, s_{288})$. However, now we will store not only all linear but also all pair quadratic equations for each value of e. Hence for each e we need to do more Trivium steps (in symbolical computing) for both the proper inner state IS_{t_0} and for the fault inner state IS'_{t_0}.

As common in cryptology, to simplify the computations we work in the factor ring $\mathrm{GF}(2)[s_1, \ldots, s_{288}]/(s_1^2 - s_1, \ldots, s_{288}^2 - s_{288})$ instead of the whole polynomial ring $\mathrm{GF}(2)[s_1, \ldots, s_{288}]$. This can be done, since for any $x \in \mathrm{GF}(2)$ we have $x^2 = x$ and we would like to make these computations as simple as possible. In this way we also obtain more equations of small degrees, since this factorisation reduces any term of type s_i^n to the term s_i.

In our implementation, we have used the value $n = 280$, so we need to obtain 280 bits of a keystream and we need to do 280 steps of Trivium (in symbolical computing) during the precomputation. We will not theoretically discuss the complexity of these precomputations, but in our implementation the precomputation phase with 280 Trivium step took couple of hours on a standard desktop computer.

The average number of equations of a degree up to 7 given by the keystream difference for a single fault injection and for a certain number of Trivium steps can be found at Tab. 2. Table 4 describes number of pair quadratic equations given by the keystream difference for a single fault injection and certain number of Trivium steps and it compares this number to the amount of all quadratic equations obtained. As we can see, the loss is around 20%.

During this attack, we store the equations obtained in a matrix M over GF(2) which has 288+287+1 columns. The first 288 columns will represent the variables $s_1, \ldots, s_{288}$ and the following 287 columns will represent all pair quadratic terms. We denote the variable of a column j by y_j and define

$$y_j = \begin{cases} s_j, & 1 \leq j \leq 288 \\ s_{j-288} \cdot s_{j-287}, & 289 \leq j \leq 575. \end{cases}$$

The last column contains the right-hand-side value for each equation. At the beginning of the attack, we put all linear and pair quadratic equations obtained

Table 4. The average number (among all fault positions) of pair quadratic equations obtained from a random fault compared to the average number of all quadratic equations

number of steps	avg. num. of all quad. eq.	avg. num. of pair quad. eq.	loss (percentage)
160	0.24	0.19	18.84%
180	1.17	0.94	19.64%
200	2.52	1.98	21.63%
220	4.14	3.19	22.75%
240	5.99	4.75	20.75%
260	7.76	6.08	21.66%
280	9.22	7.14	22.52%

from the proper keystream into M. Afterwards for each fault injection, we make fault position determination and according to the fault position, we insert the precomputed equations for the actual fault position.

In addition to the previous attack, we also hold a list of already known variables. This list will help us employ quadratic connections between variables y_i. Strictly speaking, anytime we determine the value of some previously unknown variable y_i, for some $1 \leq i \leq 288$ (so $y_i = s_i$), we go through the whole matrix M and we eliminate variables y_{i+287} and y_{i+288} in each row (only y_{289} for $i = 1$ and only y_{575} for $i = 288$). If we for example determine that for some $1 \leq i \leq 288$, $y_i = s_i = 1$, then we go through all rows of M with non-zero value in column y_{i+287} or y_{i+288}, set this variable to zero and add 1 to y_{i-1} or y_{i+1} respectively. In the case of $y_i = 0$ for some $1 \leq i \leq 288$, we just set y_{i+287} and y_{i+288} to zero. In this way, we can possibly obtain some more linear equations in $s_1, \ldots, s_{288}$ or even determine some new variables. If this is the case, we repeat this procedure again. For the rest of the paper, we denote this procedure as *quadratic_to_linear()*.

In the description of the attack, we suppose that the list of known variables is updated automatically, so we do not mention this explicitly. E.g. 2 new variables will be added automatically to the list of known variables after almost each fault injection.

During the attack, we also use classical linear algebra techniques for solving a system of linear equations in y_i, $1 \leq i \leq 575$, represented by M. Let us denote this by *elimination()*. It is clear, that this procedure can also reveal some new variables. If this happens, we use these new variables for the *quadratic_to_linear()* procedure and if this changes at least one equation in M, we do the *elimination()* again.

The attack is described by Alg. 4.

We have not theoretically analysed the complexity of this algorithm, but its running time on a standard desktop computer was always only a couple of seconds.

The average number of fault injections needed to obtain a certain number of inner states bits by the described attack is shown on Tab. 5 and illustrated on the right of the Fig. 1. These experimental results show, that the behaviour of Alg. 4 is opposite to the behaviour of Alg. 3. In this case, if we would like to obtain only 100 inner state bits, we need to inject approx. 40 faults and for 288

Algorithm 4. Attack using pair equations

```
Input:  Trivium stream cipher with possibility of fault injections (see Sect. 5.1)
Output: Secret key $K$
 1: obtain $n$ consecutive bits of $\{z_i\}$
 2: insert equations for $\{z_i\}_{i=1}^{148}$ to $M$, using bits of $\{z_i\}$ as RHS
 3: while not (all $s_1, \dots, s_{288}$ are known) do
 4:    insert a fault into $IS_{t_0}$
 5:    obtain $n$ consecutive bits of the faulty keystream $\{z'_i\}$
 6:    compute keystream difference $d_i = z'_i + z_i,\ 1 \leq i \leq n$
 7:    determine the fault position, $e$
 8:    insert equations for the keystream difference (according to value of $e$) into $M$
 9:    repeat
10:       do quadratic_to_linear()
11:    until it keeps changing $M$
12:    do elimination()
13:    if new variables obtained by elimination() then
14:       goto 9
15:    end if
16: end while
17: run Trivium backwards starting with $IS = (s_1, \dots, s_{288})$ until an inner state $IS_0 =$
    $(s_1^0, \dots, s_{288}^0)$ of type (1) (see page 162) is reached
18: output $K = (s_1^0, \dots, s_{80}^0)$.
```

Table 5. Number of fault injections needed (m) to obtain a certain number of inner state bits (T) (average over 1000 experiments) by Alg. 4

T	20	40	60	80	100	120	140	160	180	200	220	240	260	280	288
m	10.1	20.3	28.4	35.4	39.8	41.9	42.4	42.5	42.7	42.8	42.9	43.0	43.1	43.1	43.1

inner state bits we need only approx. 43 faults. It follows that stopping Alg. 4 earlier and guessing the remaining variables would be of no significance.

5.6 Possible Extensions, Future Work

In this section we will shortly describe some possible extensions of the previous attack.

The first extension could be an algorithm, which would use all quadratic equations and not only pair equations. The size of the matrix M would be much higher in this case. However, since the matrix would be sparse, it could be represented and handled efficiently. According to Tab. 4, we would get approx. 2 more equations from each fault, so this would reduce the number of faults needed to less than 40.

Next possible extension would be an attack, which would also use equations of higher order. This doesn't necessarily mean that we would try to solve systems of polynomial equations. We could only store these equations, and then use a function similar to the *quadratic_to_linear()* to eliminate terms of higher order.

E.g. if we will decide to use equations up to degree 3, we could possible eliminate cubic terms and get some new equations of degree 2. E.g. for fault position 95 we have

$$\begin{aligned} d_{256} &= s_{96}s_{81}s_{82} + s_{96}s_{56} + s_{96}s_{83} + s_{96}s_{161} + s_{96}s_{98} + s_{96}s_{97} + s_{96}s_{185} = \\ &= s_{96} \cdot (s_{81}s_{82} + s_{56} + s_{83} + s_{161} + s_{98} + s_{97} + s_{185}) \end{aligned}$$

so if $s_{96} = 1$ we get new a pair equation. In this way we could obtain more equations, which could be used in the previous attack. This would further reduce the number of fault injections needed.

By adjusting the prerequisites, we can obtain other improvements. E.g. if an attacker can choose the fault position, the number of fault injections needed for the proposed attacks would significantly reduce. Yet another option could be injection of more faults at once. It is clear, that in this case an attacker would obtain much more information from each fault injection (e. g. if two faults are injected at once, 4 inner state bits are obtained directly). Hence, an attack could be carried out using only few fault injection. On the other hand, the fault position determination could be problematic.

6 Conclusion

In this paper, differential fault analysis of Trivium was described. As far as we know, this was the first time differential fault analysis was applied to a stream cipher based on non-linear shift registers.

We have shown, that an attacker is able to obtain the secret key after approximately 43 fault injections using one of the described algorithms, assuming the chosen-ciphertext attack scenario. All the methods proposed in this article arise directly from the Trivium analysis, they are of low complexity and are easy to implement.

Acknowledgement

The authors would like to thank Martin Hlaváč for his helpful comments and the anonymous reviewers of the article for their careful readings and valuable suggestions.

References

1. De Cannière, C., Preneel, B.: Trivium: A Stream Cipher Construction Inspired by Block Cipher Design Principles. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/30 (2005), http://www.ecrypt.eu.org/stream
2. Raddum, H.: Cryptanalytic Results on Trivium. eSTREAM, ECRYPT Stream Cipher Project, Report 2006/039 (2006), http://www.ecrypt.eu.org/stream
3. Maximov, A., Biryukov, A.: Two Trivial Attacks on Trivium. eSTREAM, ECRYPT Stream Cipher Project, Report 2007/006 (2007), http://www.ecrypt.eu.org/stream

4. Babbage, S.: Some Thoughts on Trivium. eSTREAM, ECRYPT Stream Cipher Project, Report 2007/007 (2007), http://www.ecrypt.eu.org/stream
5. Turan, M.S., Kara, O.: Linear Approximations for 2-round Trivium. eSTREAM, ECRYPT Stream Cipher Project, Report 2007/008 (2007), http://www.ecrypt.eu.org/stream
6. Biham, E., Dunkelman, O.: Differential Cryptanalysis in Stream Ciphers. COSIC internal report (2007)
7. Rechberger, C., Oswald, E.: Stream Ciphers and Side-Channel Analysis. In: SASC 2004 - The State of the Art of Stream Ciphers, Workshop Record, pp. 320-326 (2004), http://www.ecrypt.eu.org/stream
8. Fischer, W., Gammel, B.M., Kniffler, O., Velten, J.: Differential Power Analysis of Stream Ciphers. eSTREAM, ECRYPT Stream Cipher Project, Report 2007/014 (2007), http://www.ecrypt.eu.org/stream
9. Hoch, J.J., Shamir, A.: Fault Analysis of Stream Ciphers. In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 240–253. Springer, Heidelberg (2004)
10. Biham, E., Granboulan, L., Nguyen, P.: Impossible Fault Analysis of RC4 and Differential Fault Analysis of RC4. In: SASC 2004 - The State of the Art of Stream Ciphers, Workshop Record, pp. 147–155 (2004), http://www.ecrypt.eu.org/stream
11. Courtois, N., Klimov, A., Patarin, J., Shamir, A.: Efficient Algorithms for Solving Overdefined Systems of Multivariate Polynomial Equations. In: Preneel, B. (ed.) EUROCRYPT 2000. LNCS, vol. 1807, pp. 392–407. Springer, Heidelberg (2000)

Appendix

Table 6. Non-zero elements of $\{d_j\}$, with $s'_i = s_i + 1$, for $1 \leq i \leq 93$

fault pos.	keystream difference d_j for $j =$												
$s'_i = s_i + 1$	67-i	94-i	136-i	151-i	161-i	162-i	163-i	176-i	177-i	178-i	202-i	220-i	242-i
$i = 1$	1	1		1	s_{i+1}	X		s_{i+1}	X	1	1		
$i = 2, \ldots, 66$	1	1		1	s_{i+1}	s_{i-1}		s_{i+1}	s_{i-1}	1	1		
$i = 67, \ldots, 69$		1	1		s_{i+1}	s_{i-1}		s_{i+1}	s_{i-1}	1		1	
$i = 70, \ldots, 90$		1			s_{i+1}	s_{i-1}	1	s_{i+1}	s_{i-1}	1			1
$i = 91$		1			s_{i+1}	s_{i-1}	1	s_{i+1}	s_{i-1}	1			
$i = 92$		1				s_{i-1}	1		s_{i-1}	1			
$i = 93$		1					1			1			

Table 7. Non-zero elements of $\{d_j\}$, with $s'_i = s_i + 1$, for $94 \leq i \leq 177$

fault pos.	keystream difference d_j for $j =$											
$s'_i = s_i + 1$	163-i	178-i	229-i	241-i	242-i	243-i	244-i	256-i	274-i	287-i	288-i	289-i
$i = 94$	1	1	1	1	s_{i+1}	X	1	1	1	s_{i+1}	X	1
$i = 95, \ldots, 162$	1	1	1	1	s_{i+1}	s_{i-1}	1	1	1	s_{i+1}	s_{i-1}	1
$i = 163, \ldots, 171$		1		1	s_{i+1}	s_{i-1}	1	1		s_{i+1}	s_{i-1}	1
$i = 172, \ldots, 174$		1			s_{i+1}	s_{i-1}	1			s_{i+1}	s_{i-1}	1
$i = 175$		1			s_{i+1}	s_{i-1}	1			s_{i+1}	s_{i-1}	1
$i = 176$		1				s_{i-1}	1				s_{i-1}	1
$i = 177$		1					1					1

Table 8. Non-zero elements of $\{d_j\}$, with $s'_i = s_i + 1$, for $178 \le i \le 288$

fault pos.	keystream difference d_j for $j =$											
$s'_i = s_i + 1$	289-i	310-i	331-i	371-i	353-i	354-i	355-i	376-i	380-i	381-i	382-i	394-i
$i = 178$	1	1	1	1	s_{i+1}	X	1	1	s_{i+1}	X	1	1
$i = 179, \ldots, 243$	1	1	1	1	s_{i+1}	s_{i-1}	1	1	s_{i+1}	s_{i-1}	1	1
$i = 244, \ldots, 264$	1		1		s_{i+1}	s_{i-1}	1	1	s_{i+1}	s_{i-1}	1	
$i = 265, \ldots, 285$	1				s_{i+1}	s_{i-1}	1		s_{i+1}	s_{i-1}	1	
$i = 286$	1				s_{i+1}	s_{i-1}	1		s_{i+1}	s_{i-1}	1	
$i = 287$	1					s_{i-1}	1			s_{i-1}	1	
$i = 288$	1						1				1	

Accelerating the Whirlpool Hash Function Using Parallel Table Lookup and Fast Cyclical Permutation

Yedidya Hilewitz[1], Yiqun Lisa Yin[2], and Ruby B. Lee[1]

[1] Department of Electrical Engineering,
Princeton University, Princeton NJ 08544, USA
{hilewitz,rblee}@princeton.edu
[2] Independent Security Consultant
yiqun@alum.mit.edu

Abstract. Hash functions are an important building block in almost all security applications. In the past few years, there have been major advances in the cryptanalysis of hash functions, especially the MDx family, and it has become important to select new hash functions for next-generation security applications. One of the potential candidates is Whirlpool, an AES-based hash function. Whirlpool adopts a very different design approach from MDx, and hence it has withstood all the latest attacks. However, its slow software performance has made it less attractive for practical use. In this paper, we present a new software implementation of Whirlpool that is significantly faster than previous ones. Our optimization leverages new ISA extensions, in particularly Parallel Table Lookup (PTLU), which has previously been proposed to accelerate block ciphers like AES and DES, multimedia and other applications. We also show a novel cyclical permutation algorithm that can concurrently convert rows of a matrix to diagonals. We obtain a speedup of 8.8× and 13.9× over a basic RISC architecture using 64-bit and 128-bit PTLU modules, respectively. This is equivalent to rates of 11.4 and 7.2 cycles/byte, respectively, which makes our Whirlpool implementation faster than the fastest published rate of 12 cycles/byte for SHA-2 in software.

1 Introduction

Hash functions form an important component in almost all security applications, e.g., digital signature schemes, to ensure the authenticity and integrity of data. Some of the most popular hash functions are MD5 [20] and SHA-1 [4]. Both have been widely deployed in practice and adopted by major security standards such as SSL/TLS and IPsec.

In the past few years, there have been major breakthroughs in the cryptanalysis of hash functions. New collision attacks on MD5 [24] and SHA-1 [23] have demonstrated serious weaknesses in their design. Built upon these attacks, researchers have also developed new attacks on hash-based security protocols such

K. Nyberg (Ed.): FSE 2008, LNCS 5086, pp. 173–188, 2008.

as X.509 digital certificate protocol [22]. While practical impact of these attacks is still debatable, it is obvious that new hash functions are needed. Indeed, NIST has already hosted two hash function workshops and has started an AES-like competition to select an Advanced Hash Standard (AHS) [17].

Whirlpool [1] is a hash function designed by Barreto and Rijmen in 2000. It is designed based on the AES with very similar structure and basic operations. It has been adopted by the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC) as part of the joint ISO/IEC international standard.

Since its publication, there have been some studies on fast implementation of Whirlpool [12,19], mostly in hardware. A comprehensive comparative study on hash function implementation [16] shows that Whirlpool is several times slower than MD5 or SHA-1 in software. Due to its relative slow performance and the prevalence of MD5 and SHA-1 in existing implementations, Whirlpool has not attracted too much attention for practical use.

With the emergence of new hash proposals, there is some renewed interest in Whirlpool. Compared with most of the new proposals, Whirlpool stands out with its AES-based clean design. Its design approach is very different from the MDx hash family, and hence may resist existing attacks that are applicable to MDx. Also, since AES is now the NIST standard for block ciphers, there is intense interest in faster implementations of AES and its security analysis. Whirlpool's similarity to AES can leverage these fast implementation techniques and facilitate its security analysis.

In this paper, we present a new software implementation of Whirlpool that is significantly faster than previous implementations. Our optimization method takes advantage of the heavy use of table lookups and byte-oriented operations in Whirlpool by leveraging processor ISA (Instruction Set Architecture) extensions that are tailored to such operations. In particular, the Parallel Table Lookup module (PTLU) [6,11] is a natural fit for the Whirlpool computation steps, thereby providing major speedup. In addition, a subword permutation instruction called `check` [14] is also useful for accelerating cyclical permutations, giving further performance enhancements. These ISA extensions have been defined previously for other purposes such as multimedia and cryptographic processing. Besides general-purpose microprocessors, these operations are even more suitable for crypto-processors and hardware ASIC (Application Specific Integrated Circuit) implementations, for fast software and hardware implementations of Whirlpool.

Our software implementation of Whirlpool attains a speedup of 8.8× with a 64-bit PTLU module and 13.9× with a 128-bit PTLU module, compared with a baseline single-issue processor. These performance results show that ISA extensions are much faster - with significantly simpler hardware - than using conventional micro-architectural performance-enhancing techniques such as superscalar execution. For example, 4-way superscalar execution achieves a speedup of only 3.3×. We also compare our Whirlpool performance with other 512-bit hash functions like SHA-2; we have a rate of 11.4 cycles/byte for the 64-bit PTLU module and 7.2 cycles/byte for the 128-bit PTLU module, while the best reported rate for SHA-2 is 12 cycles/byte on Intel Core 2 and AMD Opteron processors [21]. This suggests that Whirlpool is a viable hash function choice, providing excellent security and excellent performance.

We remark that the use of the PTLU functional unit provides not only major performance advantages but also security advantages in preventing side-channel attacks. A new concern with software implementations of cryptographic algorithms based on table lookups is the leakage of the secret key due to cache-based *software* side channel attacks, which do not require additional equipment like power or timing *physical* side channel attacks. Our proposed fast implementation of Whirlpool, when it is used in keyed hash mode, is free from such cache-based software side channel attacks.

The rest of the paper is organized as follows. Section 2 provides a high-level overview of Whirlpool. Section 3 provides the motivation for our fast implementation of Whirlpool. Section 4 describes the parallel table lookup module and other ISA extensions. Section 5 explains how to use these ISA extensions to accelerate Whirlpool. Section 6 presents performance results and Section 7 considers security advantages. Section 8 is the conclusion.

2 Whirlpool

2.1 Algorithm Overview

Like most hash functions, Whirlpool operates by iterating a compression function that has fixed-size input and output. Its compression function is a dedicated AES-like block cipher that takes a 512-bit hash state M and a 512-bit key K. (Hence, both the state and the key can be conveniently represented as 8×8 matrices with byte entries.) The iteration process adopts the well-known Miyaguchi-Preneel construction [15].

In what follows, we provide a concise description of the compression function that is most relevant to our implementation. Technical details of the algorithm can be found in [1]. At a high level, each execution of the compression function can be divided into two parts:

a. expanding the initial key K into ten 512-bit round keys, and
b. updating the hash state M by mixing M and the round keys.

Part b consists of ten rounds, and each round consists of the following four steps (labeled W1 through W4 below) with byte-oriented operations:

W1. Non-linear substitution. Each byte in the state matrix M is substituted by another byte according to a predefined substitution, $S(x)$ (aka S-box).

W2. Cyclical permutation. Each column of the state matrix M is cyclic shifted so that column j is shifted downwards by j positions.

W3. Linear diffusion. The state matrix M is multiplied with a predefined 8×8 MDS matrix C.

W4. Addition of keys. Each byte of the round key is exclusive-or'ed (XOR) to each byte of the state.

The key expansion (part a) is almost the same as the above state update, except that the initial key K is treated as the state and some pre-defined constants as the key. Hence, both parts consist of ten similar rounds.

Note that Whirlpool differs from AES in that the rounds operate on 512-bit inputs rather than 128-bit inputs. Because of the larger block size, the design of the S-box and MDS matrix is also adjusted accordingly, but the general design philosophy remains the same.

2.2 A Useful Observation by the Designers

In [1], the designers of Whirlpool suggested a method to implement each round of the compression function using only table lookup and XOR operations on a 64-bit processor. We exploit this in our optimization.

Their idea is to first define a set of tables which combine the computation of the S-box S and MDS matrix C. For $0 \le k \le 7$, let C_k be the k-th row of the MDS-matrix C. Define eight tables of the following form:

$$T_k(x) = S(x) \cdot C_k,\ 0 \le k \le 7. \tag{1}$$

Note that each table T_k has $2^8 = 256$ entries, indexed by the input x. For each x, the entry $S(x)\cdot C_k$ has eight bytes (by multiplying $S(x)$ with each of the eight bytes in the row C_k) . Hence, each table T_k is 2^{11} bytes, and the total storage is 2^{14} bytes (16 KB) for the eight tables. Given these tables, one can rewrite the operations in Steps W1 through W3 as follows. Let $M_{i,j}$ denote the (i,j)th byte in the state matrix before Step W1, and let M'_i denote the ith row in the state matrix after Step W3. Then M'_i (which is 8 bytes) can be computed as

$$M'_i = \bigoplus_{k=0}^{7} T_k(M_{(i-k) \bmod 8,k}). \tag{2}$$

For example, the first output row M'_0 can be computed as

$$\begin{aligned} M'_0 = T_0(M_{0,0}) \oplus T_1(M_{7,1}) \oplus T_2(M_{6,2}) \oplus T_3(M_{5,3}) \oplus \\ T_4(M_{4,4}) \oplus T_5(M_{3,5}) \oplus T_6(M_{2,6}) \oplus T_7(M_{1,7}). \end{aligned} \tag{3}$$

Equation (3) produces the first row of the updated state matrix M'. It is repeated to generate all 8 rows of the new state matrix, M'_i, for $i = 0, 1, \dots, 7$.

3 Motivation for Our Fast Implementation

How fast can a software implementation of Whirlpool be? Considering Equation (3), each row of the updated matrix M' can be computed with 8 selections of byte-elements of the current 8×8 matrix M, 8 table lookup operations using these 8 selected bytes as indices, and 7 exclusive-or operations. Hence, this computation takes $8d+8+7$ instructions, where d is the number of instructions needed to select a byte and place it in a register in a form that can be used by the next instruction for a load instruction (to perform the table lookup). In a typical RISC processor, $d = 3$ instructions: shift target byte to correct position, mask byte, and add to base address of table. Since Equation (3) is repeated for each of 8 rows, the number of instructions required is $8 \times (8d + 8 + 7) = 8 \times 39 = 312$ instructions. An additional 8 exclusive-or instructions are required for key addition, for a total of 320 instructions. Since this is performed for both the state and key matrices, the total number of instructions per round is $2 \times 320 = 640$ instructions.

Since the only serial dependences are between generating the index for a table lookup, doing the table lookup, then combining this result with other results using an XOR, can we achieve a faster software implementation with appropriate new instruction primitives? By appropriate instruction primitives, we mean instructions that are reasonable in cost, and have general-purpose usage for a variety of applications. Reasonable cost also suggests that any new instruction should fit the datapath structure of general-purpose microprocessors, which implies that an instruction can have at most 2 source registers and 1 result register.

Equation (3) can also be described in two steps to generate each new row of the state matrix, M':

A1. Cyclical Permutation. Select all the 8 byte-elements in parallel, placing them in the appropriate order in a register. (Step W2)

A2. Substitution and Diffusion. Look up 8 tables in parallel, using the bytes in the register generated in step A1 as indices, and immediately combine these 8 results into a single result using an XOR tree. (Steps W1 and W3)

Fig. 1. Main steps in our optimized Whirlpool software implementation

Suppose Step A1 takes x instructions and Step A2 takes y instructions. Then, the total number of instructions taken for 8 rows, for the state and key matrices, is:

$$2 \times 8 \times (x + y). \tag{4}$$

Note that $x = 24$ instructions and $y = 15$ instructions in the above calculations for the basic RISC processor.

With the microprocessor datapath restriction described above where *an instruction can have at most 2 source registers and 1 result register*, Step A1 would require $x = 4$ instructions since it needs to read from 8 different registers. Step A2 could potentially be done in $y = 1$ instruction since it has only one operand and one result. It turns out that we can indeed achieve step A2 in $y = 1$ instruction using a powerful parallel table lookup instruction (Section 4). We can do better in Step A1 using effectively only $x = 3$ instructions rather than 4, by cyclically permuting all 8 rows of the matrix concurrently (Section 5).

4 ISA Extensions

Whirlpools's heavy use of table lookup and byte-oriented computations motivate us to pay special attention to ISA extensions that are related to such operations.

We describe a parallel table lookup instruction (Section 4.1) and a subword permutation instruction (Section 4.2) previously proposed to accelerate multimedia, block ciphers and other applications.

In general, ISAs are extended when new applications emerge that require a set of operations that are not well supported by existing instructions. Emulation of these operations can take many tens or hundreds of existing instructions. Consequently, new instructions are added to perform the operations, yielding significant acceleration and, typically, reduced power consumption. For microprocessors, the goal is that the new operations are "general-purpose", meaning that they are useful in other applications beyond the initial motivating ones - the more applications the more likely the new operation will be supported in future generations of microprocessors. We show that two previously proposed operations are also useful for Whirlpool.

4.1 Parallel Table Lookup

Parallel table lookup was initially proposed to speed up block cipher execution, including AES [6], and other block ciphers including DES, 3DES, Mars, Twofish and RC4 [5]. It has also been used for fingerprinting and erasure codes to accelerate storage backup [11] and other algorithms that can employ table-lookup as an optimization.

An n-bit Parallel Table Lookup (PTLU) module consists of $n/8$ blocks of memory, each with its own read port. Fig. 2 shows a 64-bit PTLU with 8 parallel memory blocks. (A 128-bit PTLU will have 16 parallel memory blocks.) The inputs to the module are sourced from two general-purpose registers and the output is written to a single general-purpose register - hence fitting into the typical 2-source, 1-result datapath of processors. The $n/8$ blocks of memory are configured as a set of 256-entry tables, indexed by the $n/8$ bytes of the first source operand. The tables are read in parallel and the outputs from the tables are combined using a simple combinational logic function - a tree of XOR-Multiplexers (termed XMUXes). The result is then XORed with the second source operand and written to the result register.

The PTLU module is read using the following instruction:

```
ptrd.x1 r1, r2, r3
```

The bytes of r_2 are used as indices into the set of tables in the PTLU module, the outputs of which are XORed together into one value and then XORed with r_3 before being written to r_1. While a parallel table lookup only needs one source register, r_2, to supply the table indices, a second source register is available in processor datapaths, and so the XOR (or some other combination) with r_3 is essentially free in the above `ptrd.x1` instruction.

In the PTLU module proposed in [5,6,11], the XMUX's can also perform other operations like logical OR, or select the left (or right) input, in addition to the XOR operation. The "`x1`" in the `ptrd` instruction specifies that the XOR operation is selected and one 64-bit result is produced. (An "`x2`" subop is used for a 128-bit PTLU module to indicate that two 64-bit results are produced in the final XMUX stage.)

In [6], a fast instruction for loading the 8 tables in parallel is also proposed. A row across all 8 tables can be written from the contents of a data cache line in a single `ptw` instruction. Hence, only 256 instructions are needed to load 8 tables each having 256 entries, rather than 8 × 256 instructions. In [11], addressing multiple sets of tables is also described, to allow concurrent processing of different algorithms which use the parallel lookup tables, without the need for re-loading tables.

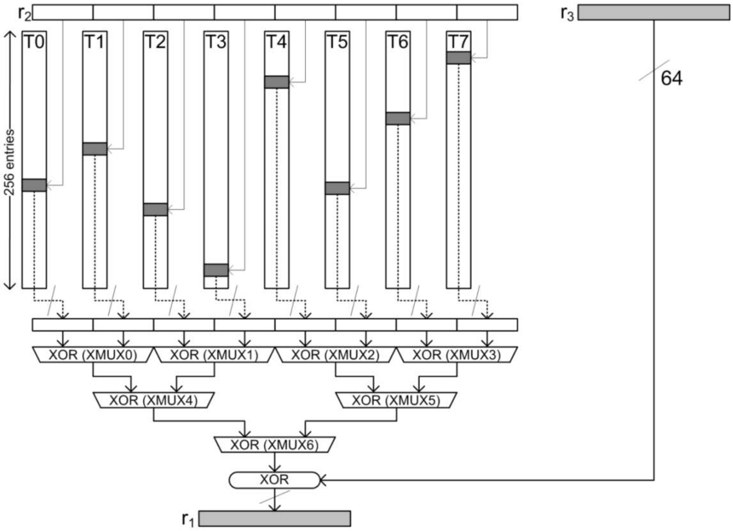


Fig. 2. PTLU module

4.2 Byte Permutations

Multimedia applications often require operations on subwords, or data smaller than the processor word (or register) size, down to a byte. ISAs have been extended with instructions that perform standard arithmetic or logical operations on these subwords in parallel as well as with instructions to efficiently rearrange these subwords in a register and between registers [13,14]. For example, the `check` instruction was defined by Lee [14] as one of a small set of subword permutation instructions for rearranging the elements of matrices in processing two-dimensional multimedia data like images, graphics and video. We propose re-using this to accelerate Whirlpool. The `check` instruction is defined as follows:

```
check.sw r1, r2, r3
```

The subwords of size sw bytes are selected alternately from the two source registers, r_2 and r_3, in a checkerboard pattern, and the result is written to r_1. In Fig. 3, each register is shown as 8 bytes, and the `check` instruction is shown for

2-byte subwords. The IBM AltiVec `vsel` instruction [9], which, for each bit position, conditionally selects from the bits of the two source operands depending on the value of the bit in a third source operand, can also be used to perform `check` when executed with the appropriate masks in the third operand. Similarly, the Intel SSE4 `pblend` instructions [10], which conditionally select subwords from two operands depending on the value of an immediate or a fixed third source operand, can also be used to perform `check`.

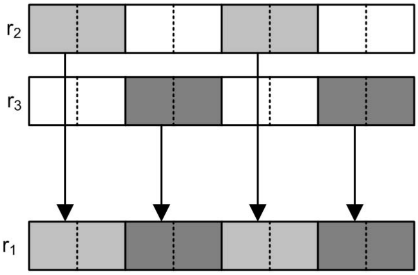


Fig. 3. check.2 r1, r2, r3 (for 64-bit registers)

5 Fast Software Implementation of Whirlpool

We now show in detail how we use the two instructions defined in Section 4 to implement the two steps in our optimized Whirlpool algorithm shown in Fig. 1. We will focus on 64-bit processors - the same techniques can be easily extended to processors with 128-bit registers, with minor variations.

Fig. 4 shows our optimized pseudocode for one round of the state update of the Whirlpool compression function on a 64-bit processor using PTLU. The 64 bytes of key are held in 8 general purpose registers (RK0-RK7) and the 64 bytes of state are held in 8 general purpose registers (RM0-RM7). The eight PTLU tables contain the eight tables from Equation (1), which combine steps W1 and W3 (Section 2) of the Whirlpool algorithm, also labeled step *A2* (Section 3) . A further optimization with PTLU is that step W4 is also combined with steps W1 and W3 by a single `ptrd` instruction. Step W2, also labeled step *A1* (Section 3), is performed in the Cyclical Permute function described in Section 5.2.

5.1 Using PTLU for Substitution and Diffusion (Step A2)

A single PTLU read instruction updates a row of the state matrix, performing the eight table lookups of Equation (2) at once. For example, the instruction

```
ptrd.x1 RM0, RM0, RK0
```

corresponds to Equation (3), which details the state transformation of row 0. The eight bytes in row 0 of M: $M_{0,0}, M_{7,1}, \ldots, M_{1,7}$, are stored in RM0 after the cyclical permutation step. These 8 bytes are used as the indices into the set of eight

```
# RM0-RM7 are the 8 state registers
# RK0-RK7 are the 8 key registers

Cyclical_Permute(RM0-RM7)
ptrd.x1 RM0, RM0, RK0
ptrd.x1 RM1, RM1, RK1
ptrd.x1 RM2, RM2, RK2
ptrd.x1 RM3, RM3, RK3
ptrd.x1 RM4, RM4, RK4
ptrd.x1 RM5, RM5, RK5
ptrd.x1 RM6, RM6, RK6
ptrd.x1 RM7, RM7, RK7
```

Fig. 4. Pseudocode for one round of the state update of Whirlpool compression

R0	00	01	02	03	04	05	06	07
R1	10	11	12	13	14	15	16	17
R2	20	21	22	23	24	25	26	27
R3	30	31	32	33	34	35	36	37
R4	40	41	42	43	44	45	46	47
R5	50	51	52	53	54	55	56	57
R6	60	61	62	63	64	65	66	67
R7	70	71	72	73	74	75	76	77

(a)

R0'	00	71	62	53	44	35	26	17
R1'	10	01	72	63	54	45	36	27
R2'	20	11	02	73	64	55	46	37
R3'	30	21	12	03	74	65	56	47
R4'	40	31	22	13	04	75	66	57
R5'	50	41	32	23	14	05	76	67
R6'	60	51	42	33	24	15	06	77
R7'	70	61	52	43	34	25	16	07

(b)

Fig. 5. (a) 8 × 8 matrix at start of round; (b) 8 × 8 matrix after cyclical permutation

tables defined by Equation (1) which are stored in the PTLU module (Fig. 2). The eight 64-bit table entries read out, $T_0(M_{0,0}), T_1(M_{7,1}), \ldots, T_7(M_{1,7})$, are XORed together by the XMUX tree. At this point, the PTLU module has performed Equation (3). The output of the XMUX tree is also XORed with the first row of the key matrix stored in RK0, completing the state transformation of row 0. The updated row 0 of M is then written back to RM0. Seven more `ptrd` instructions update the remaining 7 rows of M.

5.2 Novel Algorithm for Cyclical Permutation

The state matrix at the start of a round is shown in Fig. 5(a). The transformed matrix, used in the table lookup, is shown in Fig. 5(b). This transformation is the columnar cyclical permutation of the Whirlpool compression function, accomplished by rotating the jth column down by j positions. We propose a novel algorithm that accomplishes this in a logarithmic number of steps. First, move columns 1, 3, 5 and 7 down by 1 row. Second, move columns 2 and 3, 6 and 7 down by 2 rows. At this point, columns 0 and 4 have been moved down by 0 rows, columns 1 and 5 by 1 row, columns 2 and 6 by 2 rows, and columns 3 and 7 by 3 rows. Third, move columns 4, 5, 6 and 7 down by 4 rows. This achieves the desired result, where column j has been moved down by j rows.

R0	00	01	02	03	04	05	06	07
R7	70	71	72	73	74	75	76	77

⇓

R0’	00	71	02	73	04	75	06	77

Fig. 6. check.1 R0’, R0, R7

The transformation by cyclical permutation from Fig. 5(a) to Fig. 5(b) turns rows of the matrix into (wrapped) diagonals. In [14], Lee showed how two `check.1` instructions can be used to rotate one column of each 2 × 2 matrix mapped across 2 registers. We propose using the `check.sw` instructions, doubling the subword size (`sw`) at each step, to turn the eight rows of the 8 × 8 matrix of bytes (in eight 64-bit registers) into eight diagonals.

First, we execute a `check.1` instruction on each row and its neighbor one row above (Fig. 6), which selects one byte alternately from the two registers. This has the effect of rotating columns 1, 3, 5 and 7 down by one position (Fig. 7(a)). Second, we execute a `check.2` instruction on each row and its neighbor two rows above, which selects 2 bytes alternately from the two registers. This has the effect of rotating columns 2, 3, 6 and 7 down by an additional two positions (Fig. 7(b)). Third, we execute a `check.4` instruction on each row and its neighbor four rows above, which selects 4 bytes alternately from the two registers. This results in rotating columns 4, 5, 6 and 7 down an additional four positions to yield the final permutation (Fig. 5(b)).

R0’	00	71	02	73	04	75	06	77
R1’	10	01	12	03	14	05	16	07
R2’	20	11	22	13	24	15	26	17
R3’	30	21	32	23	34	25	36	27
R4’	40	31	42	33	44	35	46	37
R5’	50	41	52	43	54	45	56	47
R6’	60	51	62	53	64	55	66	57
R7’	70	61	72	63	74	65	76	67

(a)

R0	00	71	62	53	04	75	66	57
R1	10	01	72	63	14	05	76	67
R2	20	11	02	73	24	15	06	77
R3	30	21	12	03	34	25	16	07
R4	40	31	22	13	44	35	26	17
R5	50	41	32	23	54	45	36	27
R6	60	51	42	33	64	55	46	37
R7	70	61	52	43	74	65	56	47

(b)

Fig. 7. (a) State matrix with columns 1, 3, 5 and 7 rotated down by 1 position; (b) State matrix with columns 1 and 5 rotated down by one position, columns 2 and 6 by two positions and columns 3 and 7 by three positions

5.3 Register Usage and Instruction Counts

Register usage: Most RISC processors have only 32 General Purpose Registers. Our software implementation requires only 24 registers, 8 each for key, state and scratch space, plus a few registers for memory pointers. The first step of the cyclical permutation writes its result to 8 scratch registers, the second step writes back to the original 8 registers, the third step writes to the scratch registers, and the ptrd instruction writes back to the original registers. Thus our implementation is not constrained by register allocation.

Instruction Counts: Updating the state matrix takes 32 instructions total as $8 \times \lg(8) = 24$ check instructions are needed to cyclically permute the matrix and 8 ptrd instructions are needed to complete the update (see Fig. 4). The key matrix undergoes a similar update with the only difference being an additional load instruction to retrieve the round constant. Thus one round of the Whirlpool compression function takes 65 instructions with PTLU-64.

Without PTLU, a round takes approximately 640 instructions on a basic RISC processor (Section 3). Thus, using PTLU reduces the instruction count by an order of magnitude. In Section 6, we consider cycle counts of the full Whirlpool hash function.

5.4 Extending the Techniques to PTLU-128

For a processor with 128-bit registers, a PTLU-128 module with 16 parallel memory blocks can be used (Fig. 2 shows PTLU-64 with 8 memory blocks). In a PTLU-128 version of the parallel lookup instruction, ptrd.x2, the 16 bytes of the first source register are used as indices into the 16 tables, the outputs of which are XORed into 2 parallel 64-bit values, which are each XORed with the second source register before being written to the 128-bit destination register.

For 128-bit registers, the cyclical permutation step also requires an instruction to rearrange the bytes within a word, as two rows are contained within a single processor register. We use a byteperm instruction, also defined in [6]. In this instruction, the first source register holds the data to be permuted and the second source register lists the new ordering for the bytes of the data. This instruction is similar to the IBM AltiVec vperm instruction [9] or the IA-32 pshufb instruction [10] and is only needed for the 128-bit PTLU module, not for the 64-bit PTLU module.

In total, 8 check instructions and 8 byteperm instructions are needed to cyclically permute the matrix (held in only 4 128-bit registers) in a 128-bit processor; the precise sequence of instructions is omitted for brevity. Only 4 ptrd.x2 instructions are needed for each of the key and state matrix transformations in a round as two iterations of Equation (2) are done in parallel with ptrd.x2. Hence, a Whirlpool round takes only $2 \times 20 + 1 = 41$ instructions with a 128-bit PTLU.

Commodity microprocessors have 128-bit register files for their multimedia instructions like SSE for Intel x86 processors [10] and AltiVec for PowerPC processors [9]. Hence, it is not unreasonable to add a 128-bit PTLU unit to the multimedia functional units using the 128-bit registers already present.

6 Performance Analysis

Table 1 summarizes the performance improvement for Whirlpool over the basic 64-bit RISC processor for single-issue 64-bit and 128-bit processors with PTLU

Table 1. Relative Performance of Whirlpool

baseline	Speedup with Superscalar			Speedup with PTLU, 1-way	
	2-way	4-way	8-way	64-bit	128-bit
1	1.65	3.26	5.97	8.79	13.90

Table 2. Performance of Whirlpool and SHA-2

Algorithm	Processor	Cycles per Byte
Whirlpool	**PTLU-64**	11.41
	PTLU-128	7.22
	Pentium III (asm)	36.52 [16]
	Core 2 (C)	44 [21]
	Opteron (C)	38 [21]
SHA-2 512	Pentium III (asm)	40.18 [16]
	Core 2 (C)	12 [21]
	Opteron (C)	12 [21] / 13.4 [8]

and for 64-bit superscalar execution (evaluated using the SimpleScalar Alpha simulator [2]). We compare our performance using ISA extensions to superscalar execution, because the latter is the technique typically used by processor designers to increase performance by executing multiple instructions each processor cycle. k-way superscalar means the execution of k instructions per cycle. In general, the hardware cost of superscalar execution increases exponentially with k, while the performance increases less than linearly with k.

While Whirlpool scales well with superscalar execution, ranging from 1.65× to 5.97× for 2-way to 8-way superscalar, adding a PTLU module (and using the `check` and `byteperm` instructions) yields even better results: 8.79× with a 64-bit PTLU and 13.90× with a 128-bit PTLU. The latter can be compared to the 1.65× speedup of a 2-way superscalar processor, as both perform the equivalent of two instructions per cycle - the processor with 128-bit PTLU is 8.42× faster. Even the 64-bit PTLU with 1 instruction per cycle is faster than the very complex 8-way superscalar processor.

In Table 2, we compare our results (PTLU-64 and PTLU-128) with the performance of Whirlpool on some existing processors [16,21], and with the performance of the SHA-2 512-bit hash function [16,21,8]. The single-issue 64-bit processor with PTLU greatly outperforms more complex 3- and 4-way superscalar processors like the AMD Opteron or the Intel Core 2.

We also estimated the performance of Whirlpool on the Intel Core 2 hypothetically enhanced with a single PTLU-128 module using its 16 128-bit SSE registers. The performance result is slightly slower than that of our single issue RISC processor with PTLU-128. This is due to the Core 2 machine having a `byteperm` (implemented by `pshufb`) with a 3 cycle latency and 2 cycle pipelined instruction issue. (Note that later Core 2 processors have a “super shuffle engine” with a 1 cycle `pshufb`.) Performance was also impacted by extra copy instructions due to IA-32 instructions overwriting one of the source operands, and limited superscalar speedup due to the single PTLU module and serialization restrictions on the byte permutation instructions. Nevertheless, due to the tremendous performance boost provided by the PTLU-128 module, our Whirlpool implementation still has better performance than SHA-2 on the complex Intel Core 2 microprocessor.

7 Security Advantages

In Section 1, we discussed the security advantages of the Whirlpool algorithm, in light of recent advances in finding collisions in MD-5 and SHA-1 hash functions. We now discuss the additional advantages of using PTLU in our Whirlpool implementation in thwarting side-channel attacks as well.

Cache side-channel timing attacks [18] have recently been shown to be viable against cryptographic algorithms that use lookup tables stored in cache, such as AES. One such attack forces part of the lookup table out of the cache and then measures the time of a subsequent encryption. If the encryption takes longer than the baseline time, it implies that the part of the table that was evicted from the cache had to be refetched from main memory. This provides information about the key bytes. The general idea can also be applied to keyed hash functions that use lookup tables.

Using PTLU to perform the table lookups precludes these timing attacks from taking place, as the tables do not reside in cache. Table access time is always a constant for all tables in the PTLU module. Multiple processes can use the same Whirlpool PTLU tables without impacting each other. If another process needs the PTLU module, either multiple sets of tables may be implemented in hardware or the OS is responsible for fully replacing and restoring the table contents during context switch. Consequently, the use of PTLU for Whirlpool not only provides tremendous performance improvements but also increases the security of the implementation when Whirlpool is used in keyed mode such as for MACs.

In general, the use of PTLU can protect crypto algorithms from cache-based side-channel attacks. This would allow table lookup to continue to be an effective *non-linear component* in ciphers and hash functions. For the MDx hash family, the linear relation between the hash state and the input message has proved to be a major weakness that made these functions vulnerable to the so-called message modification techniques [23,24]. Whirlpool, with its heavy use of table lookup, provides excellent resistance against this line of new attacks on hash functions.

8 Conclusions

We have presented a fast software implementation of the hash function Whirlpool, based on ISA extensions that permit parallel table lookup and a novel algorithm that performs the cyclical permutation of the columns of the state (or key) matrix in parallel. We show that the PTLU (parallel table lookup) module, together with `check`, a subword permutation instruction, can greatly improve the performance of Whirlpool. More specifically, on a single-issue 64-bit processor, our software implementation provides an 8.79× speedup, more than the 5.97× speedup gained from the much more complex hardware technique of 8-way superscalar execution. With our speedup, Whirlpool is faster than SHA-512, both of which produce 512-bit hash results.

Our optimization approach is somewhat different from existing ones. While most research in fast software implementations has focused on how to optimize given existing ISA, we also try to address the problem from the other direction. That is, what ISA extensions are most useful to speed up existing algorithms? The ISA extensions used in our implementation have already been defined and applied earlier to accelerate multimedia and cryptographic processing. Our new results on Whirlpool, together with the earlier work, support the inclusion of more powerful ISA extensions in both general-purpose processors and crypto-processors. In particular, the fact that many crypto algorithms make heavy use of table lookups make the PTLU module and associated instructions very attractive for future CPUs. Additionally, the use of PTLU inoculates these crypto algorithms against cache-based software side channel attacks.

Due to Whirlpool's initial performance problem, its designers have proposed the Maelstrom-0 hash function [7] as a replacement. This new hash function changes the key schedule, but uses the same compression function for updating the hash state. Consequently, the techniques presented in this paper will speed up Maelstrom-0 as well.

Designing and selecting new hash functions is a hot subject for both the crypto research community and the security industry. Our new implementation results suggest that, in addition to its security, Whirlpool can also have great performance. Therefore, Whirlpool can be a viable hash function choice for next-generation security applications.

Acknowledgments. Y. Hilewitz is supported by NSF and Hertz Foundation Fellowships.

References

1. Barreto, P.S.L.M., Rijmen, V.: The Whirlpool Hashing Function, `http://paginas.terra.com.br/informatica/paulobarreto/WhirlpoolPage.html`
2. Burger, D., Austin, T.: The SimpleScalar Tool Set, Version 2.0. University of Wisconsin-Madison Computer Sciences Department Technical Report #1342 (1997)
3. CACTI 4.2. HP Labs, `http://www.hpl.hp.com/personal/Norman_Jouppi/cacti4.html`
4. Federal Information Processing Standards (FIPS) Publication 180-1. Secure Hash Standard (SHS). U.S. DoC/NIST (1995)
5. Fiskiran, A.M.: Instruction Set Architecture for Accelerating Cryptographic Processing in Wireless Computing Devices. PhD Thesis, Princeton University (2005)

6. Fiskiran, A.M., Lee, R.B.: On-Chip Lookup Tables for Fast Symmetric-Key Encryption. In: Proceedings of the IEEE 16th International Conference on Application-Specific Systems, Architectures and Processors (ASAP), pp. 356–363. IEEE, Los Alamitos (2005)
7. Gazzoni Filho, D.L., Barreto, P.S.L.M., Rijmen, V.: The Maelstrom-0 Hash Function. In: VI Brazilian Symposium on Information and Computer Systems Security (2006)
8. Gladman, B.: SHA1, SHA2, HMAC and Key Derivation in C, http://fp.gladman.plus.com/cryptography_technology/sha/index.htm
9. IBM Corporation. PowerPC Microprocessor Family: AltiVec Technology Programming Environments Manual. Version 2.0 (2003)
10. Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual, vol. 1-2 (2007)
11. Josephson, W., Lee, R.B., Li, K.: ISA Support for Fingerprinting and Erasure Codes. In: Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP). IEEE Computer Society Press, Los Alamitos (2007)
12. Kitsos, P., Koufopavlou, O.: Whirlpool Hash Function: Architecture and VLSI Implementation. In: Proceedings of the 2004 International Symposium on Circuits and Systems (ISCAS 2004), pp. 23–36 (2004)
13. Lee, R.B.: Subword Parallelism with MAX-2. IEEE Micro. 16(4), 51–59 (1996)
14. Lee, R.B.: Subword Permutation Instructions for Two-Dimensional Multimedia Processing in MicroSIMD Architectures. In: Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors, pp. 3–14. IEEE Computer Society Press, Los Alamitos (2000)
15. Menezes, A., van Orschot, P., Vanstone, S.: Handbook of applied cryptography. CRC Press, Boca Raton (1997)
16. Nakajima, J., Matsui, M.: Performance Analysis and Parallel Implementation of Dedicated Hash Functions. In: Knudsen, L.R. (ed.) EUROCRYPT 2002. LNCS, vol. 2332, pp. 165–180. Springer, Heidelberg (2002)
17. NIST. Hash Function Main Page, http://www.nist.gov/hash-competition
18. Osvik, D.A., Shamir, A., Tromer, E.: Cache Attacks and Countermeasures: the Case of AES. Cryptology ePrint Archive, Report 2005/271 (2005)
19. Pramstaller, N., Rechberger, C., Rijmen, V.: A Compact FPGA Implementation of the Hash Function Whirlpool. In: Proceedings of 14th International Symposium on Field Programmable Gate Arrays, pp. 159–166 (2006)
20. Rivest, R.L.: The MD5 message-digest algorithm. Request for comments (RFC) 1321, Internet Activities Board, Internet Privacy Task Force (1992)
21. St. Denis, T.: LibTomCrypt Benchmarks, http://libtomcrypt.com/ltc113.html
22. Stevens, M., Lenstra, A., de Weger, B.: Chosen-prefix Collisions for MD5 and Colliding X. In: Naor, M. (ed.) EUROCRYPT 2007. LNCS, vol. 4515, pp. 1–22. Springer, Heidelberg (2007)
23. Wang, X., Yin, Y.L., Yu, H.: Finding Collisions in the Full SHA-1. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 17–36. Springer, Heidelberg (2005)
24. Wang, X., Yu, H.: How to Break MD5 and Other Hash Functions. In: Cramer, R.J.F. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 19–35. Springer, Heidelberg (2005)

Appendix A: Hardware Cost Analysis

We estimate the cost in terms of area and latency of adding PTLU, `byteperm` and `check`. For PTLU we used CACTI [3] to estimate the latency and area of the tables and we synthesized the XMUX tree using a TSMC 90nm library. We compare the access time latency and area of our 64-bit PTLU module with a cache of the same capacity (i.e., 16 Kilobyte cache), and also compare our 128-bit PTLU with a 32 Kilobyte cache. The 64-bit PTLU module, which has 16KB of tables, has 88% of the latency and 92% of the area of a 16KB 2-way associative cache with 64 byte lines. The 128-bit PTLU module has 75% of the latency and 79% of the area of a 32KB 2-way associative cache with 64 byte lines. In each case, we find that the PTLU module is faster and smaller than a typical data cache of the same capacity. Still, the two modules have larger latencies than an ALU, so we conservatively estimate the `ptrd` instruction to take two processor cycles. Since the results of the table lookups are not needed right away (Fig. 4), this has no impact on performance.

For `byteperm` and `check`, in an ISA such as IA-32 or IA-64 that has a multimedia subword permutation unit, the cost of adding these instructions, if they do not already exist, is negligible. For other ISAs, support for the `byteperm` instruction can be added to the shifter unit with minimal impact to area and without affecting the cycle time [6]. The `check` instruction can be implemented by a set of $n/8$ 8-bit 2:1 multiplexers with the control bit pattern selected from a small set of fixed bitstrings: $(0^k1^k)^{n/2k}$, $k = 1, 2, 4, \ldots$ and n the register width in bytes. Thus, it can also be easily added without area or cycle time impact.

Appendix B: Related Work

Byte permutation instructions such as the `byteperm` instruction described (or the PowerPC AltiVec `vperm` [9] or Intel SSSE3 `pshufb` [10] mentioned above), can be used as a limited PTLU instruction. For example, in the `vperm` instruction, which uses three 128-bit registers, the bytes of the third source operand are indices that select bytes in the first two source operands. The latter can be considered a single 32-entry table, with byte entries. With `byteperm` or `pshufb`, which only have 2 source registers, the first operand functions as a 16-entry table. These instructions can be used for the S-box non-linear substitutions, which map a byte to a byte, in AES or Whirlpool implementations that explicitly perform all four steps (W1, W2, W3, W4) of the state transformation (Section 2). However, the PTLU instruction used in this paper is much more capable.

Second Preimage Attack on 3-Pass HAVAL and Partial Key-Recovery Attacks on HMAC/NMAC-3-Pass HAVAL

Eunjin Lee[1], Donghoon Chang[1], Jongsung Kim[1], Jaechul Sung[2], and Seokhie Hong[1]

[1] Center for Information Security Technologies(CIST),
Korea University, Seoul, Korea
{walgadak,pointchang,joshep,hsh}@cist.korea.ac.kr
[2] University of Seoul,Seoul, Korea
jcsung@uos.ac.kr

Abstract. In 1992, Zheng, Pieprzyk and Seberry proposed a one-way hashing algorithm called HAVAL, which compresses a message of arbitrary length into a digest of 128, 160, 192, 224 or 256 bits. It operates in so called passes where each pass contains 32 steps. The number of passes can be chosen equal to 3, 4 or 5. In this paper, we devise a new differential path of 3-pass HAVAL with probability 2^{-114}, which allows us to design a second preimage attack on 3-pass HAVAL and partial key recovery attacks on HMAC/NMAC-3-pass HAVAL. Our partial key-recovery attack works with 2^{122} oracle queries, $5 \cdot 2^{32}$ memory bytes and 2^{96} 3-pass HAVAL computations.

Keywords: HAVAL, NMAC, HMAC, Second preimage attack, Key recovery attack.

1 Introduction

In 2004 and 2005, Biham *et al.* and Wang *et al.* published several important cryptanalytic articles [1,2,12,13,14,15] that demonstrate efficient collision search algorithms for the MD4-family of hash functions. Their proposed neutral-bit and message modification techniques make it possible to significantly improve previous known collision attacks on MD4, MD5, HAVAL, RIPEMD, SHA-0 and SHA-1 [3,9,10,17], including the second preimage attack on MD4 which finds a second preimage for a random message with probability 2^{-56} [18].

There have also been several articles that present attacks on NMAC and HMAC based on the MD4 family. In 2006, Kim *et al.* first proposed distinguishing and forgery attacks on NMAC and HMAC based on the full or reduced HAVAL, MD4, MD5, SHA-0 and SHA-1 [7] and Contini and Yin presented forgery and partial key recovery attacks on HMAC/NMAC-MD4, -SHA-0, -reduced 34-round SHA-1 and NMAC-MD5 [4]. More recently, full key-recovery attacks on HMAC/NMAC-MD4, reduced 61-round SHA-1 and NMAC-MD5 were proposed in FC 2007 [8] and in CRYPTO 2007 [6].

K. Nyberg (Ed.): FSE 2008, LNCS 5086, pp. 189–206, 2008.

The motivation of this paper is that 1) there are strong collision producing differentials of HAVAL for collision attacks [10,11], but no differential of HAVAL has been proposed for second preimage attacks, and 2) there are distinguishing/forgery attacks on HMAC/NMAC-HAVAL [7], but no key-recovery attack has been proposed. This paper investigates if 3-pass HAVAL and HMAC/NMAC-3-pass HAVAL are vulnerable to the second preimage and partial key recovery attacks, respectively. (After our submission, we learned that Hongbo Yu worked independently for her doctoral dissertation [16] on partial key recovery attacks on HAVAL-based HMAC and second preimage attack on HAVAL).

The cryptographic hash function HAVAL was proposed by Y. Zheng et al. in 1992 [19]. It takes an input value of arbitrary length and digests it into variant lengths of 128, 160, 192, 224 or 256 bits. In this paper, we present a new second preimage differential path of 3-pass HAVAL with probability 2^{-114} and devise a second preimage attack on 3-pass HAVAL, and a partial key recovery attack on HMAC/NMAC-3-pass HAVAL with 2^{122} oracle queries, $5 \cdot 2^{32}$ memory bytes and 2^{96} 3-pass HAVAL computations.

This paper is organized as follows. In Section 2, we describe HAVAL, HMAC, NMAC, and notations. Next, we present a second preimage attack on 3-pass HAVAL in Section 3 and apply it to recover a partial key of HMAC/NMAC-3-pass HAVAL in Section 4. Finally, we conclude in Section 5.

2 Preliminaries

In this section, we give a brief description of the HAVAL hash function, the HMAC/NMAC algorithms and notations used in the paper.

2.1 Description of HAVAL

HAVAL produces hashes in different lengths of 128, 160, 192, 224 and 256 bits. It allows that users can choose the number of passes 3, 4 or 5, where each pass contains 32 steps. It computes the hashes in the following procedure:

- Padding: an inserted message is padded into a multiple of 1024 bits.
- Compression function H: let $M^0, M^1, \cdots, M^S$ be 1024-bit message blocks and each M^i consists of 32 32-bit words, that is, $M^i = M^i_0||M^i_1||\cdots||\, M^i_{31}$, where M^i_j is a 32-bit word.
  - $h_0 = H(IV, M^0)$, where IV is the initial value.
  - $h_1 = H(h_0, M^1), \cdots, h_s = H(h_{s-1}, M^S)$
- Output of HAVAL: H_n

The HAVAL compression function H processes 3, 4 or 5 passes. Let F_1, F_2, F_3, F_4 and F_5 be the five passes and (D_{in}, M) be the input value of H, where D_{in} is a 256-bit initial block and M is a 1024-bit message block. Then the output of the compression function D_{out} can be computed in the following way.

$$E_0 = D_{in},\ E_1 = F_1(E_0, M),\ E_2 = F_2(E_1, M),\ E_3 = F_3(E_2, M);$$
$$E_4 = F_4(E_3, M)\ \ (pass = 4, 5),\ E_5 = F_5(E_4, M)\ \ (pass = 5);$$

$$D_{out} = \begin{cases} E_3 \boxplus E_0, & pass = 3 \\ E_4 \boxplus E_0, & pass = 4 \\ E_5 \boxplus E_0, & pass = 5 \end{cases}$$

Fig. 1 shows the i-th step of HAVAL, where a_i represents the updated 32-bit value of the i-th step. Let a 1024-bit message block M be denoted $M = M_0||M_1||\cdots||M_{30}||M_{31}$, where M_i $(i = 0, 1, \cdots, 31)$ is a 32-bit word, then the orders of the message words in each pass are as in Table 1.

Each pass employs a different Boolean function f_i $(i = 1, 2, 3, 4, 5)$ and a different permutation function. The following f_i is used in pass i:

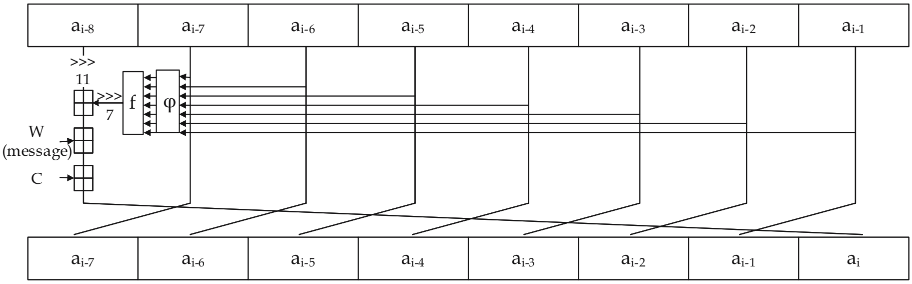


Fig. 1. i-th step of HAVAL hash function

Table 1. Orders of message words

$Pass1$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$Pass2$	5	14	26	18	11	28	7	16	0	23	20	22	1	10	4	8
	30	3	21	9	17	24	29	6	19	12	15	13	2	25	31	27
$Pass3$	19	9	4	20	28	17	8	22	29	14	25	12	24	30	16	26
	31	15	7	3	1	0	28	27	13	6	21	10	23	11	5	2
$Pass4$	24	4	0	14	2	7	28	23	26	6	30	20	18	25	19	3
	22	11	31	21	8	27	12	9	1	29	5	15	17	10	16	13
$Pass5$	27	3	21	26	17	11	20	29	19	0	12	7	13	8	31	10
	5	9	14	30	18	6	28	24	2	23	16	22	4	1	25	15

$$\begin{aligned}
f_1(x_6, x_5, x_4, x_3, x_2, x_1, x_0) &= x_1x_4 \oplus x_2x_5 \oplus x_3x_6 \oplus x_0x_1 \oplus x_0 \\
f_2(x_6, x_5, x_4, x_3, x_2, x_1, x_0) &= x_1x_2x_3 \oplus x_2x_4x_5 \oplus x_1x_2 \oplus x_1x_4 \oplus \\
&\quad\; x_2x_6 \oplus x_3x_5 \oplus x_4x_5 \oplus x_0x_2 \oplus x_0 \\
f_3(x_6, x_5, x_4, x_3, x_2, x_1, x_0) &= x_1x_2x_3 \oplus x_1x_4 \oplus x_2x_5 \oplus x_3x_6 \oplus x_0x_3 \oplus x_0 \\
f_4(x_6, x_5, x_4, x_3, x_2, x_1, x_0) &= x_1x_2x_3 \oplus x_2x_4x_5 \oplus x_3x_4x_6 \oplus x_1x_4 \oplus x_2x_6 \oplus \\
&\quad\; x_3x_4 \oplus x_3x_5 \oplus x_3x_6 \oplus x_4x_5 \oplus x_4x_6 \oplus x_0x_4 \oplus x_0 \\
f_5(x_6, x_5, x_4, x_3, x_2, x_1, x_0) &= x_1x_4 \oplus x_2x_5 \oplus x_3x_6 \oplus x_0x_1x_2x_3 \oplus x_0x_5 \oplus x_0
\end{aligned}$$

Let $\varphi_{i,j}$ be the permutation function of the j-th pass of the i-pass HAVAL. Table 2 shows the $\varphi_{i,j}$ used in each pass. In each step, the updated value a_i is computed as

$$a_i = (a_{i-8} \ggg 11) \boxplus (f(\varphi(a_{i-7}, a_{i-6}, \cdots, a_{i-1})) \ggg 7) \boxplus M_i \boxplus C,$$

where $X \ggg i$ is the right cyclic rotation of X by i bits, and C is a constant.

Table 2. $\varphi_{i,j}$ used in each pass

permutations	$x_6\ x_5\ x_4\ x_3\ x_2\ x_1\ x_0$
$\varphi_{3,1}$	$x_1\ x_0\ x_3\ x_5\ x_6\ x_2\ x_4$
$\varphi_{3,2}$	$x_4\ x_2\ x_1\ x_0\ x_5\ x_3\ x_6$
$\varphi_{3,3}$	$x_6\ x_1\ x_2\ x_3\ x_4\ x_5\ x_0$
$\varphi_{4,1}$	$x_2\ x_6\ x_1\ x_4\ x_5\ x_3\ x_0$
$\varphi_{4,2}$	$x_3\ x_5\ x_2\ x_0\ x_1\ x_6\ x_4$
$\varphi_{4,3}$	$x_1\ x_4\ x_3\ x_6\ x_0\ x_2\ x_5$
$\varphi_{4,4}$	$x_6\ x_4\ x_0\ x_5\ x_2\ x_1\ x_3$
$\varphi_{5,1}$	$x_3\ x_4\ x_1\ x_0\ x_5\ x_2\ x_6$
$\varphi_{5,2}$	$x_6\ x_2\ x_1\ x_0\ x_3\ x_4\ x_5$
$\varphi_{5,3}$	$x_2\ x_6\ x_0\ x_4\ x_3\ x_1\ x_5$
$\varphi_{5,4}$	$x_1\ x_5\ x_3\ x_2\ x_0\ x_4\ x_6$
$\varphi_{5,5}$	$x_2\ x_5\ x_0\ x_6\ x_4\ x_3\ x_1$

2.2 Description of HMAC/NMAC

Fig. 2 shows NMAC and HMAC based on a compression function f which maps $\{0,1\}^n \times \{0,1\}^b$ to $\{0,1\}^n$. The K_1 and K_2 are all n-bit keys and the $\overline{K} = K||0^{b-n}$, where K is an n-bit key. The opad is formed by repeating the byte '$\mathsf{0x36}$' as many times as needed to get a b-bit block, and the ipad is defined similarly using the byte '$\mathsf{0x5c}$'.

Let $F : \{IV\} \times (\{0,1\}^b)^* \rightarrow \{0,1\}^n$ be the iterated hash function defined as $F(IV, M^1||M^2||\cdots||M^S) = f(\cdots f(f(IV, M^1), M^2)\cdots, M^S)$, where M^i is a b bit message. Let g be a padding method, $g(x) = x||10^t||\mathsf{bin}_{64}(x)$, where t is the smallest non-negative integer such that $g(x)$ is a multiple of b and $\mathsf{bin}_i(x)$ is the i-bit binary representation of x. Then, NMAC and HMAC are defined as follows:

$$\begin{aligned} \mathrm{NMAC}_{K_1,K_2}(M) &= H(K_2, g(H(K_1, g(M)))) \\ \mathrm{HMAC}_K(M) &= H(IV, g(\overline{K} \oplus \mathsf{opad}||H(IV, g(\overline{K} \oplus \mathsf{ipad}||M)))). \end{aligned}$$

2.3 Notations

Let M and M' be 1024-bit messages such that $M = M_0||M_1||\cdots||M_{31}$ and $M' = M'_0||M'_1||\cdots||M'_{31}$, where M_i $(i = 0, 1, 2, \cdots, 31)$ and M'_j $(j = 0, 1, 2, \cdots, 31)$ are 32-bit words. We denote by a_i (resp., a'_i) the updated value of the i-th step using

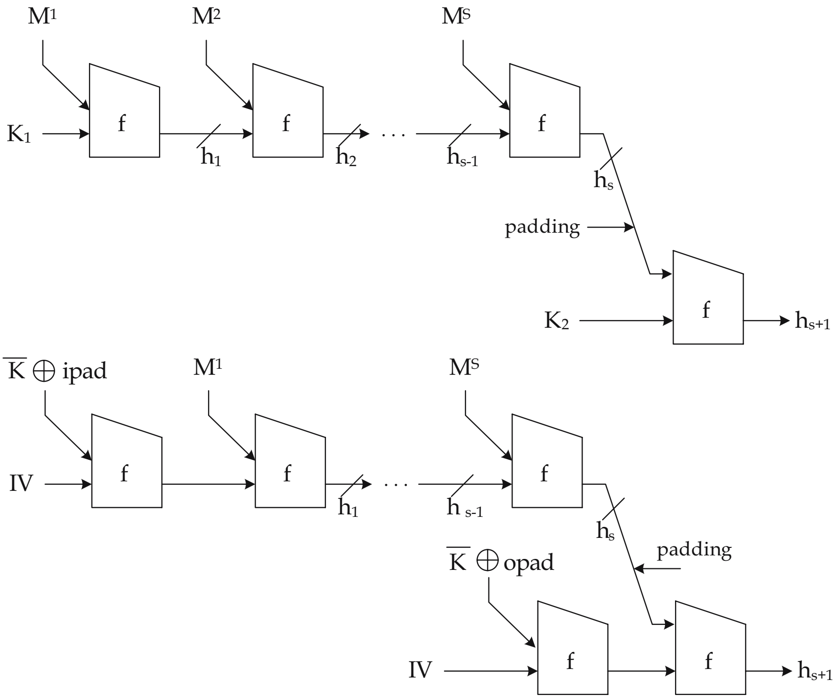


Fig. 2. NMAC and HMAC

the message M (resp., M'). Let t_i (resp., t'_i) be the output value of the Boolean function of the i-th step using the message M (resp., M'). The j-th bits of a_i and t_i are denoted $a_{i,j}$ and $t_{i,j}$. Additionally, we use several following notations in our attacks, where $0 \leq j \leq 31$.

- $a_i[j]$: $a_{i,j} = 0, a'_{i,j} = 1$,
- $a_i[-j]$: $a_{i,j} = 1, a'_{i,j} = 0$,
- $t_i[j]$: $t_{i,j} = 0, t'_{i,j} = 1$,
- $t_i[-j]$: $t_{i,j} = 1, t'_{i,j} = 0$.

3 Second Preimage Attack on 3-Pass HAVAL

In this section, we show how to construct a second preimage differential path of 3-pass HAVAL. Using this differential path, we find a second preimage of 3-pass HAVAL with probability 2^{-114}, i.e., for a given message M, we find another message M' with probability 2^{-114} satisfying $H(M) = H(M')$, where H is 3-pass HAVAL. Our differential path of 3-pass HAVAL is stronger than the previous ones [7,9,11,12] against the second preimage attack.

3.1 Second Preimage Differential Path of 3-Pass HAVAL

Let two 1024-bit message blocks $M = M_0|| M_1||M_2|| \cdots ||M_{31}$ and $M' = M'_0||M'_1|| M'_2|| \cdots ||M'_{31}$ satisfy $M_i = M'_i$ for $i = 0, 1, \cdots , 21, 23, 24, \cdots , 31$ and $M_{22} \oplus M'_{22} = 2^{31}$. Then we can use these two messages to construct a second preimage differential path of 3-pass HAVAL with probability 2^{-114}. Table 3 shows our second preimage differential path of 3-pass HAVAL, which has been constructed as follows.

First of all, from the message pair we get the input difference to the 23-rd step $(\Delta a_{15}, \Delta a_{16}, \Delta a_{17}, \Delta a_{18}, \Delta a_{19}, \Delta a_{20}, \Delta a_{21}, \Delta a_{22}) = (0, 0, 0, 0, 0, 0, 0, a_{22}[31])$ if a condition $a_{22,31} = 0$ holds. Recall that $(a_{i-8}, a_{i-7}, \cdots , a_{i-2}, a_{i-1})$ is the input state to the i-th step. We assume that the output differences of the Boolean functions from the 23-rd step to the 36-th step are all zeroes. Then we can obtain the input difference to the 37-th step is $(0, a_{30}[20], 0, 0, 0, 0, 0, 0)$. It is easy to see that the required assumption works if several conditions hold in our differential, which we call sufficient conditions. For example, consider a difference Δt_{24}. The input difference to the 24-th step is $(\Delta a_{16}, \Delta a_{17}, \Delta a_{18}, \Delta a_{19}, \Delta a_{20}, \Delta a_{21}, \Delta a_{22}, \Delta a_{23}) = (0, 0, 0, 0, 0, 0, a_{22}[31], 0)$. The permutation is $\varphi(x_6, x_5, x_4, x_3, x_2, x_1, x_0) = (x_1, x_0, x_3, x_5, x_6, x_2, x_4)$ and the Boolean function is $f(x_6, x_5, x_4, x_3, x_2, x_1, x_0) = x_1x_4 \oplus x_2x_5 \oplus x_3x_6 \oplus x_0x_1 \oplus x_0$ in the 24-th step. Thus, $f(\varphi(x_6, x_5, x_4, x_3, x_2, x_1, x_0)) = x_2x_3 \oplus x_6x_0 \oplus x_5x_1 \oplus x_4x_2 \oplus x_4$ and the most significant bit of the output of the Boolean function in the 24-th step is $a_{20,31}a_{21,31} \oplus a_{17,31}a_{23,31} \oplus a_{18,31}a_{22,31} \oplus a_{19,31}a_{21,31} \oplus a_{19,31}$. If $a_{18,31} = 0$, then the difference of $a_{22,31}$ does not have effect on the output difference of the Boolean function and thus $\Delta t_{24} = 0$. Thus, $a_{18,31} = 0$ is one of the sufficient conditions. We show in Table 5 of appendix all the sufficient conditions which satisfy our differential path.

In order to compute the probability that a message M satisfies the sufficient conditions listed in Table 5, we need to check the dependency of the conditions. To make the problem easier we first solve and simplify the conditions. In this process we may reduce the number of the sufficient conditions. Consider the conditions on the 20-th bit from the 31-st step to the 37-th step in Table 5.

1. 31-st step : $a_{30,20} = 0,\ a_{24,20} = 0$
2. 32-nd step : $a_{29,20}a_{26,20} \oplus a_{28,20} \oplus a_{29,20} = 0$
3. 33-rd step : $a_{31,20}a_{27,20} \oplus a_{32,20} \oplus a_{31,20} = 0$
4. 34-th step : $a_{33,20}a_{28,20} \oplus a_{28,20} \oplus a_{32,20} = 0$
5. 35-th step : $a_{29,20} = 0$
6. 36-th step : $a_{35,20}a_{32,20} \oplus a_{34,20}a_{33,20} \oplus a_{32,20} \oplus a_{31,20} \oplus a_{35,20} = 0$
7. 37-th step : $a_{31,20} = 0,\ a_{33,20}a_{35,20} \oplus a_{36,20}a_{34,20} \oplus a_{35,20}a_{34,20} = 0$

In the 32-nd step, we can simplify the condition to $a_{28,20} = 0$ by inserting the value $a_{29,20} = 0$ which is the condition in the 35-th step. Using this condition $a_{28,20} = 0$, we can obtain $a_{32,20} = 0$ in the 34-th step. This simplified condition $a_{32,20} = 0$ and the 37-th step condition $a_{31,20} = 0$ make the 33-rd step condition always hold. Moreover, the 36-th step condition is simplified to $a_{34,20}a_{33,20} \oplus a_{35,20} = 0$ due to the conditions $a_{31,20} = 0$ and $a_{32,20} = 0$. Following is the simplified conditions for steps 31-37 (note that the number of the sufficient conditions is reduced from 9 to 8 by solving the conditions):

Table 3. Second preimage differential path of 3-pass HAVAL

step	ΔM_i	Δt_i	Δa_{i-8}	Δa_{i-7}	Δa_{i-6}	Δa_{i-5}	Δa_{i-4}	Δa_{i-3}	Δa_{i-2}	Δa_{i-1}
0	0	0	0	0	0	0	0	0	0	0
...	...	...		...					...	
21	0	0	0	0	0	0	0	0	0	0
22	±31	0	0	0	0	0	0	0	0	0
23	0	0	0	0	0	0	0	0	0	31
24	0	0	0	0	0	0	0	0	31	0
25	0	0	0	0	0	0	0	31	0	0
26	0	0	0	0	0	0	31	0	0	0
27	0	0	0	0	0	31	0	0	0	0
28	0	0	0	0	31	0	0	0	0	0
29	0	0	0	31	0	0	0	0	0	0
30	0	0	31	0	0	0	0	0	0	0
31	0	0	0	0	0	0	0	0	0	20
32	0	0	0	0	0	0	0	0	20	0
33	0	0	0	0	0	0	0	20	0	0
34	0	0	0	0	0	0	20	0	0	0
35	0	0	0	0	0	20	0	0	0	0
36	0	0	0	0	20	0	0	0	0	0
37	0	20	0	20	0	0	0	0	0	0
38	0	0	20	0	0	0	0	0	0	13
39	0	0	0	0	0	0	0	0	13	9
40	0	0	0	0	0	0	0	13	9	0
41	0	0	0	0	0	0	13	9	0	0
42	0	13	0	0	0	13	9	0	0	0
43	±31	±6	0	0	13	9	0	0	0	6
44	0	0	0	13	9	0	0	0	6	0
45	0	-9	13	9	0	0	0	6	0	0
46	0	0	9	0	0	0	6	0	0	0
47	0	0	0	0	0	6	0	0	0	30
48	0	0	0	0	6	0	0	0	30	0
49	0	0	0	6	0	0	0	30	0	0
50	0	0	6	0	0	0	30	0	0	0
51	0	0	0	0	0	30	0	0	0	-27,28
52	0	0	0	0	30	0	0	0	-27,28	0
53	0	30	0	30	0	0	0	-27,28	0	0
54	0	0	30	0	0	0	-27,28	0	0	23
55	0	28	0	0	0	-27,28	0	0	23	19
56	0	21	0	0	-27,28	0	0	23	19	21
57	0	0	0	-27,28	0	0	23	19	21	-14,15
58	0	-23	-27,28	0	0	23	19	21	-14,15	0
59	0	0	0	0	23	19	21	-14,15	0	0
60	0	0	0	23	19	21	-14,15	0	0	0
61	0	-19	23	19	21	-14,15	0	0	0	0
62	0	-15	19	21	-14,15	0	0	0	0	0
63	0	0	21	-14,15	0	0	0	0	0	0
64	0	-10	-14,15	0	0	0	0	0	0	10
65	0	0	0	0	0	0	0	0	10	0
66	0	0	0	0	0	0	0	10	0	0
67	0	0	0	0	0	0	10	0	0	0
68	0	0	0	0	0	10	0	0	0	0
69	0	0	0	0	10	0	0	0	0	0
70	0	0	0	10	0	0	0	0	0	0
71	±31	0	10	0	0	0	0	0	0	0
72	0	0	0	0	0	0	0	0	0	0
...	...	...		...					...	
95	0	0	0	0	0	0	0	0	0	0

1. 31-st step : $a_{30,20} = 0,\ a_{24,20} = 0$
2. 32-nd step : $a_{28,20} = 0$
3. 33-rd step : no condition
4. 34-th step : $a_{32,20} = 0$
5. 35-th step : $a_{29,20} = 0$
6. 36-th step : $a_{34,20}a_{33,20} \oplus a_{35,20} = 0$
7. 37-th step : $a_{31,20} = 0,\ a_{33,20}a_{35,20} \oplus a_{36,20}a_{34,20} \oplus a_{35,20}a_{34,20} = 0$

Table 6 in appendix collects all the simplified conditions for those of Table 5. We notice that the number of the sufficient conditions listed in Table 6 is 112, which seems to make the probability that a message satisfy all these conditions is 2^{-112}. However, it is not 2^{-112}, but approximately 2^{-114}. This is due to the fact that there are still dependencies in some conditions. For example, consider the conditions on the 13-th bit from the 38-th step to the 41-st step in Table 6.

1. 38-th step : $a_{38,13} = 1,\ a_{34,13}a_{32,13} \oplus a_{35,13} = 0$
2. 39-th step : $a_{33,13} \neq a_{35,13}$
3. 40-th step : $a_{34,13} \neq a_{39,13}$
4. 41-st step : $a_{40,13}a_{35,13} \oplus a_{35,13} \oplus a_{39,13} = 1$

These 5 conditions do not hold with probability 2^{-5}, but with probability $2^{-3} \cdot \frac{3}{16}$. The reason is as follows. The probability that the condition $a_{38,13} = 1$ is satisfied is 2^{-1}. Table 4 lists all the possible values of $a_{32,13}, a_{34,13}$ and $a_{35,13}$ which satisfy $a_{34,13}a_{32,13} \oplus a_{35,13} = 0$. The probability that this condition holds is $\frac{1}{2}(= \frac{4}{8})$ according to Table 4. In the 39-th step, the probability that $a_{33,13} \neq a_{35,13}$ is satisfied is 2^{-1} since $a_{33,13}$ is used only in the 39-th step. In the 40-th and 41-st steps, if $a_{35,13} = 0$, then $a_{39,13}$ and $a_{34,13}$ should be 0 and 1, respectively, and $a_{40,13}$ is either 0 or 1. The probability that $a_{35,13} = 0$ and $a_{34,13} = 1$ hold is $\frac{1}{4}$ (one out of four cases, see Table 4). Thus the probability that $a_{34,13} = 1, a_{35,13} = 0$, and $a_{39,13} = 0$ are satisfied is $\frac{1}{8}(= \frac{1}{4} \cdot \frac{1}{2})$ (recall that $a_{40,13}$ does not have effect on the condition $a_{40,13}a_{35,13} \oplus a_{35,13} \oplus a_{39,13} = 1$). If $a_{35,13} = 1$ and $a_{39,13} = 1$, then $a_{40,13} = 1$ and $a_{34,13} = 0$ due to the conditions $a_{40,13}a_{35,13} \oplus a_{35,13} \oplus a_{39,13} = 1$ and $a_{34,13} \neq a_{39,13}$. However, this is a contradiction to the condition of the 38-th step (see Table 4), and thus if $a_{35,13} = 1$, then $a_{39,13} = 0$, $a_{40,13} = 1$ and $a_{34,13} = 1$. The probability that $a_{35,13} = 1$ and $a_{34,13} = 1$ hold is $\frac{1}{4}$ by Table 4 and each probability of $a_{39,13} = 0$ and $a_{40,13} = 1$ is $\frac{1}{2}$, so the probability that $(a_{34,13}, a_{35,13}, a_{39,13}, a_{40,13}) = (1, 1, 0, 1)$ is $\frac{1}{16}$. Therefore, we can compute the probability that the conditions in the 40-th and 41-st step hold is $\frac{3}{16}(= \frac{1}{8} + \frac{1}{16})$, leading to a total probability $2^{-3} \cdot \frac{3}{16}$ for the above 5 conditions. In this way, we analyze the probability that the sufficient conditions in Table 6 are satisfied is 2^{-114}.

3.2 Attack on 3-Pass HAVAL

The second preimage resistance on a hash function plays an important role to block the attacker to produce a second preimage when a meaningful and sensitive message (e.g. a finance-related message) is used. In literature, it is defined as follows:

Table 4. Possible values for the conditions on the 38-th, 40-th and 41-st step

step		$a_{32,13}$	$a_{34,13}$	$a_{35,13}$	probability
38		1	1	1	1/8
		0	1	0	1/8
		1	0	0	1/8
		0	0	0	1/8

step	$a_{34,13}$	$a_{35,13}$	$a_{39,13}$	$a_{40,13}$	probability
40, 41	1	0	0	0	$1/4 \times 1/2 \times 1/2$
	1	0	0	1	$1/4 \times 1/2 \times 1/2$
	1	1	0	1	$1/4 \times 1/2 \times 1/2$

Second preimage resistance on a hash function H. for any given message M, it is computationally infeasible to find another message M' satisfying $H(M) = H(M')$

It follows that the second preimage attack on a hash function exists if for a given message M there is an algorithm that finds another message M' such that $H(M) = H(M')$ with probability larger than 2^{-n}, where n is the bit-length of hash values. The second preimage attack on 3-pass HAVAL works due to our differential path;

- For a given message M, the probability that M holds the sufficient conditions listed in Table 6 is 2^{-114}.
- If the message M holds the sufficient conditions, then the message M' which only differs from M at the most significant bit of the 22-nd message word has a same hash value.

4 Partial Key-Recovery Attacks on HMAC/NMAC-3-Pass HAVAL

In this section, we present partial key recovery attacks on HMAC/NMAC-3-pass HAVAL, which works based on our differential path described in Section 3. More precisely, we show how to find the partial key K_1 of NMAC-3-pass HAVAL and $f(\bar{K} \oplus \mathsf{ipad})$ of HMAC-3-pass HAVAL (note that knowing $f(\bar{K} \oplus \mathsf{ipad})$ and $f(\bar{K} \oplus \mathsf{opad})$ allows to compute the MAC value for any message). Since HMAC = NMAC if $f(\bar{K} \oplus \mathsf{ipad}) = K_1$ and $f(\bar{K} \oplus \mathsf{opad}) = K_2$, we focus on the NMAC-3-pass HAVAL attack which finds K_1 with message/MAC pairs. Recall that K_1 is placed at the position of the initial state in NMAC. This implies that recovering the initial value of 3-pass HAVAL is equivalent to getting the partial key K_1 of NMAC-3-pass HAVAL.

The main idea behind of our attack is that the attacker can recover the initial state of NMAC-3-pass HAVAL (in our attack it is K_1) if he knows a 256-bit input value at any step of 3-pass HAVAL. This idea has firstly been introduced in [4]. In this section, we first find a_{16}, a_{18}, a_{21} and a_{23} which are used as a part of an input value to the 24-th step. Remaining four-word input values a_{17}, a_{19}, a_{20}

and a_{22} to the 24-th step is then found by 2^{128} exhaustive searches. Let $a_{i,j}$ be the j-th bit of a_i and $\gamma_i = (a_{i-8} \ggg 11) \boxplus (t_i \ggg 7) \boxplus C$, where C is a constant used in step i (note $\gamma_i \boxplus M_i = a_i$).

The value a_{16} is then revealed by the following Algorithm.

Algorithm 1. In order to recover the value a_{16}, we use a condition $a_{16,31} = 0$ depicted in Table 6. The procedure goes as follows:

1. The attacker has access to the oracle $\mathcal{O}$ (=NMAC-3-pass HAVAL) and makes 2^{121} queries for 2^{120} message pairs $M = M_0, M_1, \cdots, M_{30}, M_{31}$ and $M' = M'_0, M'_1, \cdots, M'_{30}, M'_{31}$ that have the message difference given in Table 5. Among the 2^{120} message pairs, $M_0, M_1, \cdots, M_{15}$ and $M'_0, M'_1, \cdots, M'_{15}$ are all identically fixed, M_{16} and M'_{16} vary in all 2^{32} possible values, and 2^{88} message pairs in the remaining words $M_{17}, M_{18}, \cdots, M_{31}$ and $M'_{17}, M'_{18}, \cdots, M'_{31}$ are randomly chosen. In this case, what the attacker knows is that γ_{16} is identically fixed for all the 2^{120} message pairs even though he does not know the actual value γ_{16}.
2. For each candidate value γ_{16} in $0, 1, \cdots, 2^{32} - 1$;
   (a) Choose the message pairs (M, M') that make collisions for the corresponding MAC pairs.
   (b) Count the number of the message pairs chosen in Step 2(a) that satisfy $msb(\gamma_{16} \boxplus M_{16}) = 1$.
3. Output $\gamma_{16} \boxplus M_{16}$ as a_{16}, where γ_{16} has the least count number in Step 2 (b).

As mentioned before, this algorithm works due to our differential with probability 2^{-114}. Notice that our differential encompasses a sufficient condition $a_{16,31} = 0$, and each message pair among the 2^{120} message pairs satisfies the condition $a_{16,31} = 0$, our differential holds with probability 2^{-113} with respect to this message pair. If the message pair (M, M') makes the most significant bits of a_{16} and a'_{16} be 1, then the probability that the message pair (M, M') makes a collision is $2^{-121}(= 2^{-113} \cdot 2^{-8})$, for it forces additionally 8 more sufficient conditions in our collision producing differential. The reason is as follows. If $a_{16,31} = 1$, then a difference Δt_{23} is not zero, but $\pm 2^{31}$. However, this difference value can be canceled by the output difference of the Boolean function in the 31-st step. In this procedure, each of steps 24-31 requires one more additional condition, leading to total 8 additional conditions. Thus, the probability that the message pair (M, M') has a same MAC value is not a random probability but 2^{-121}, where the most significant bits of a_{16} and a'_{16} are 1. It follows that if the right γ_{16} is guessed, we expect $2^{-2}(= 2^{119} \cdot 2^{-121})$ collision pairs. On the other hand, if γ_{16} is wrongly guessed, the expectation of collision pairs is $2^5(= 2^{118} \cdot 2^{-113} + 2^{118} \cdot 2^{-121})$, (note that in the group of the message pairs such that $msb(\gamma_{16} \boxplus M_{16}) = 1$ there are on average half message pairs satisfying the actual $a_{16,31} = 0$). Since the probability that a wrong γ_{16} does not cause any collision pair is $(1 - 2^{-113})^{2^{118}} \cdot (1 - 2^{-121})^{2^{118}} < (1 - 2^{-113})^{2^{118}} (\approx e^{-32}) < 2^{-32}$, we expect that there is no wrong γ_{16} which leads to no collision in Step 2. Hence, we can determine the right γ_{16}. To summarize, Algorithm 1 requires 2^{121} oracle queries (in Step 1) and 2^{32} memory bytes (the memory complexity of this attack is dominated by the counters for γ_{16}).

Next, we show how to recover the value a_{18}, for which we use the condition $a_{18,31} = 0$ required in our differential. Since there is no condition on a_{17} (see Table 6), the attacker chooses any message word M_{17}. The main idea is similar to Algorithm 1.

First of all, the attacker selects 2^{119} message pairs (M, M') that have the message difference given in Table 6, where $M_0, M_1, \cdots, M_{17}$ and $M'_0, M'_1, \cdots, M'_{17}$ are all identically fixed ($M_0, M_1, \cdots, M_{16}$ and $M'_0, M'_1, \cdots, M'_{16}$ should be the same as those selected in Algorithm 1, which leads to $a_{16,31} = 0$), M_{18} and M'_{18} vary in all 2^{32} possible values, and 2^{87} message pairs in the remaining words are randomly chosen. Once the attacker gets the corresponding MAC pairs, he performs Steps 2 and 3 of Algorithm 1 to recover a_{18} by setting γ_{18}, M_{18}, a_{18} instead of γ_{16}, M_{16} and a_{16}. The reason why recovering a_{18} requires half of the message pairs, compared to when recovering a_{16}, is that this attack algorithm exploits message pairs satisfying $a_{16,31} = 0$ from the beginning. It increases by twice the probability that our differential holds. The remaining analysis is the same as that of Algorithm 1. To summarize, recovering a_{18} requires 2^{120} oracle queries.

Next, let us see how to recover a_{21}. In order to recover a_{21} we need to use the condition $a_{20,31} = a_{21,31}$, which is of a different form from the previous two conditions $a_{16,31} = a_{18,31} = 0$. However, the core in our attack is that $a_{20,31}$ is always a same value if $M_0, M_1, \cdots, M_{20}$ and $M'_0, M'_1, \cdots, M'_{20}$ are all identically fixed in all required message pairs, i.e, in 2^{118} message pairs (note that all these message pairs satisfy $a_{16,31} = a_{18,31} = 0$, which the attacker can select from the above algorithms). Similarly, among the 2^{118} pairs, M_{21} and M'_{21} vary in all 2^{32} possible values and 2^{86} pairs of remaining words are randomly chosen.

Algorithm 2. The attack algorithm to recover a_{21} goes as follows:

1. The attacker chooses the 2^{118} message pairs as above and asks the oracle $\mathcal{O}$ for the corresponding 2^{118} MAC pairs.
2. Choose the message pairs (M, M') that make collisions for the corresponding MAC pairs.
3. For each candidate value γ_{21} in $0, 1, \cdots, 2^{32} - 1$;
   (a) Divide two groups of which one contains message pairs that satisfy msb $(\gamma_{21} \boxplus M_{21}) = 0$ and the other one contains message pairs that satisfy $msb(\gamma_{21} \boxplus M_{21}) = 1$.
   (b) Count the number of message pairs in each group that make collisions for the corresponding MAC pairs
4. Output $\gamma_{21} \boxplus M_{21}$ as a_{21}, where γ_{21} is the value that has a group containing the least count, and M_{21} is the one of the values satisfying $a_{20,31} = a_{21,31}$.

If the values a_{20} and a_{21} satisfy the sufficient condition $a_{20,31} = a_{21,31}$, then the probability that the message pair (M, M') makes a collision is 2^{-111} (note that the three conditions $a_{16,31} = a_{18,31} = 0, a_{21,31} = a_{20,31}$ are excluded in the list of the sufficient conditions). On the other hand, if $a_{21,31} \neq a_{20,31}$, then the probability that the message pair (M, M') makes a collision is 2^{-119} (similarly, 8 more conditions are additionally needed). In case the right γ_{21} is guessed, one of the two groups is expected to have $2^{-111} \cdot 2^{117} = 2^6$ collision pairs and the other one is expected to have $2^{-119} \cdot 2^{117} = 2^{-2}$ collision pair. On the other hand, if a wrong γ is guessed, then the both groups are expected to have $2^{-111} \cdot 2^{116} = 2^5$ collision pairs each. It implies that the probability that a wrong γ_{16} does not cause any collision pair is about $e^{-32} < 2^{-32}$, and thus there is no wrong γ_{21} to pass Step 3. To summarize, Algorithm 2 needs 2^{119} oracle queries and 2^{32} memory bytes. Recovering a_{23} is quite similar to recovering a_{16} and a_{18}, which requires 2^{118} oracle queries.

Exhaustive Search for the Remaining Four Words. Using the above algorithms, we can compute the 128-bit a_{16}, a_{18}, a_{21} and a_{23} values. The remaining 128-bit a_{17}, a_{19}, a_{20} and a_{22} values are found by the following algorithm. We consider a message pair (M, M') selected from the above algorithms which makes a collision.

1. Guess a 128-bit a_{17}, a_{19}, a_{20}, a_{22} value;
   (a) Check with the computed $a_{16}, a_{18}, a_{21}, a_{23}$ and the guessed a_{17}, a_{19}, a_{20}, a_{22} that the message pair (M, M') makes a collision. If so, we determine the guessed value as the right value. Otherwise, repeat Step 1.
   (b) For the given message pair (M, M') and $a_{16}, a_{17}, \cdots, a_{22}$, recover the initial value.

If a wrong value is guessed, the probability that it causes a collision is 2^{-256}. Since the number of wrong $a_{17}, a_{19}, a_{20}, a_{22}$ tested in the attack is 2^{128} at most, we can recover the right initial value. The time complexity of the exhaustive search step is 2^{128} 3-pass HAVAL computations.

As a result, our partial key recovery attack has $2^{121} + 2^{120} + 2^{119} + 2^{118} = 2^{121.9}$ oracle queries and 2^{128} 3-pass HAVAL computations.

Reducing the Number of the 3-Pass HAVAL Computations. As described above, our partial key-recovery attack is completed by two phases; the first phase is to recover some portions of the 256-bit input value at step i, and the second is the exhaustive search phase for its remaining input bits. If we apply our attack to the input value to step 29 instead of step 24, then we can recover $a_{21}, a_{23}, a_{24}, a_{26}$ and a_{28} from the first phase with 2^{122} oracle queries and we recover the remaining a_{22}, a_{25} and a_{27} with 2^{96} 3-pass HAVAL computations from the second phase.

5 Conclusion

In this paper, we have presented a new second preimage differential path of 3-pass HAVAL with probability 2^{-114} and exploited it to devise a second preimage attack on 3-pass HAVAL, and partial key-recovery attacks on HMAC/NMAC-3-pass HAVAL with 2^{122} oracle queries, $5 \cdot 2^{32}$ memory bytes and 2^{96} 3-pass HAVAL computations. We expect that our attacks would be useful for the further analysis of HAVAL and HMAC/NMAC-HAVAL (e.g., full key-recovery attacks on HMAC/NMAC-HAVAL).

References

1. Biham, E., Chen, R.: Near-Collisions of SHA-0. In: Franklin, M. (ed.) CRYPTO 2004. LNCS, vol. 3152, pp. 290–305. Springer, Heidelberg (2004)
2. Biham, E., Chen, R., Joux, A., Carribault, P., Lemuet, C., Jalby, W.: Collisions of SHA-0 and Reduced SHA-1. In: Cramer, R.J.F. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 22–35. Springer, Heidelberg (2005)
3. Boer, B.D., Bosselaers, A.: Collisions for the Compression Function of MD-5. In: Helleseth, T. (ed.) EUROCRYPT 1993. LNCS, vol. 765, pp. 293–304. Springer, Heidelberg (1994)
4. Contini, S., Yin, Y.L.: Forgery and Partial Key-Recovery Attacks on HMAC and NMAC Using Hash Collisions. In: Lai, X., Chen, K. (eds.) ASIACRYPT 2006. LNCS, vol. 4284, pp. 37–53. Springer, Heidelberg (2006)
5. Dobbertin, H.: Cryptanalsis of MD4. In: Gollmann, D. (ed.) FSE 1996. LNCS, vol. 1039, pp. 53–69. Springer, Heidelberg (1996)
6. Fouque, P.A., Leurent, G., Nguyen, P.Q.: Full Key-Recovery Attacks on HMAC/NMAC-MD4 and NMAC-MD5. In: Menezes, A. (ed.) CRYPTO 2007. LNCS, vol. 4622, pp. 13–30. Springer, Heidelberg (2007)
7. Kim, J., Biryukov, A., Preneel, B., Hong, S.: On the Security of HMAC and NMAC Based on HAVAL, MD4, MD5, SHA-0 and SHA-1. In: De Prisco, R., Yung, M. (eds.) SCN 2006. LNCS, vol. 4116, pp. 242–256. Springer, Heidelberg (2006)
8. Rechberger, C., Rijmen, V.: On Authentication With HMAC and Non-Rondom Properties. In: Dietrich, S., Dhamija, R. (eds.) FC 2007 and USEC 2007. LNCS, vol. 4886, pp. 119–133. Springer, Heidelberg (2007)
9. Van Rompay, B., Biryukov, A., Preneel, B., Vandewalle, J.: Cryptanalysis of 3-pass HAVAL. In: Laih, C.-S. (ed.) ASIACRYPT 2003. LNCS, vol. 2894, pp. 228–245. Springer, Heidelberg (2003)
10. Wang, X., Feng, D., Lai, X., Yu, H.: Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD, Cryptology ePrint Archive, Report 2004/199 (2007)
11. Wang, X., Feng, D., Yu, H.: The Collision Attack on Hash Function HAVAL-128. Science in China, Series E 35(4), 405–416 (2005)
12. Wang, X., Lai, X., Feng, D., Chen, H., Yu, X.: Cryptanalysis of the Hash Functions MD4 and RIPEMD. In: Cramer, R.J.F. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 1–18. Springer, Heidelberg (2005)
13. Wang, X., Yin, X.Y., Yu, H.: Finding Collision in the Full SHA-1. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 17–36. Springer, Heidelberg (2005)
14. Wang, X., Yu, H., Yin, X.Y.: Efficient Collision Search Attacks on SHA-0. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 1–16. Springer, Heidelberg (2005)
15. Wang, X., Yu, H.: How to Break MD5 and Other Hash Functions. In: Cramer, R.J.F. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 19–35. Springer, Heidelberg (2005)
16. Yu, H.: Cryptanalysis of Hash Functions and HMAC/NMAC, Doctoral dissertation, SHANDONG
17. Yu, H., Wang, X., Yun, A., Park, S.: Cryptanalysis of the Full HAVAL with 4 and 5 Passes. In: Robshaw, M.J.B. (ed.) FSE 2006. LNCS, vol. 4047, pp. 89–110. Springer, Heidelberg (2006)
18. Yu, H., Wang, G., Zhang, G., Wang, X.: The Second-Preimage Attack on MD4. In: Desmedt, Y.G., Wang, H., Mu, Y., Li, Y. (eds.) CANS 2005. LNCS, vol. 3810, pp. 1–12. Springer, Heidelberg (2005)
19. Zheng, Y., Pieprzyk, J., Seberry, J.: HAVAL - a one-way hashing algorithm with variable length of output. In: Zheng, Y., Seberry, J. (eds.) AUSCRYPT 1992. LNCS, vol. 718, pp. 83–104. Springer, Heidelberg (1993)

A Sufficient Conditions of the Second Preimage Differential Path of 3-Pass HAVAL

Table 5 shows the sufficient conditions of the second preimage differential path of 3-pass HAVAL, which are derived from the property of the Boolean function f_i of appendix B. We solve and simplify the conditions of Table 5 and list the solutions in Table 6.

B Property of the Boolean Functions f_1, f_2 and f_3

Recall that the input value of the i-th step is denoted $a_{i-8}, a_{i-7}, \cdots, a_{i-1}$ and the output value of the Boolean functions of the i-th step is denoted t_i. Tables 7, 8 and 9 show the relations between the input difference and t_i of the i-th step. In the column of Assumption in the Tables, $\mathbf{a_s}[\mathbf{j}]$ represents the difference $(\Delta a_{i-8}, a_{i-7}, \cdots, \Delta a_s, \cdots, \Delta a_{i-1}) = (0, 0, \cdots, a_s[j], 0, \cdots, 0)$ for $i-1 \leq s \leq i-7$ and $t_i[]$ means that the output difference of the Boolean function of the i-th step is zero (see Section 2.3 for the notations $a_s[j]$ and $t_i[j]$). Note that even though the sign is altered from $+j$ to $-j$ in both $\mathbf{a_s}[\mathbf{j}]$ and $t_i[j]$, still the conditions are the same as in Tables 7, 8 and 9, however if the sign is altered only in one of $\mathbf{a_s}[\mathbf{j}]$ and $t_i[j]$, the second conditions should be 1 (and the first ones are not altered).

Table 5. Sufficient conditions of the second preimage differential path of 3-pass HAVAL

S	Sufficient conditions
23	$a_{16,31} = 0, a_{22,31} = 0$
24	$a_{18,31} = 0$
25	$a_{20,31} = a_{21,31}$
26	$a_{23,31} = 0$
27	$a_{24,31} = 1$
28	$a_{26,31} = 0$
29	$a_{28,31} = 0$
31	$a_{30,20} = 0,\ a_{24,20} = 0$
32	$a_{29,20}a_{26,20} \oplus a_{28,20} \oplus a_{29,20} = 0$
33	$a_{31,20}a_{27,20} \oplus a_{32,20} \oplus a_{31,20} = 0$
34	$a_{33,20}a_{28,20} \oplus a_{28,20} \oplus a_{32,20} = 0$
35	$a_{29,20} = 0$
36	$a_{35,20}a_{32,20} \oplus a_{34,20}a_{33,20} \oplus a_{32,20} \oplus a_{31,20} \oplus a_{35,20} = 0$
37	$a_{31,20} = 0,\ a_{33,20}a_{35,20} \oplus a_{36,20}a_{34,20} \oplus a_{35,20}a_{34,20} = 0$
38	$a_{37,13} = 0,\ a_{34,13}a_{32,13} \oplus a_{35,13} = 0$
39	$a_{38,9} = 0,\ a_{35,9}a_{33,9} \oplus a_{36,9} = 0,\ a_{36,13}a_{33,13} \oplus a_{35,13} \oplus a_{36,13} = 0$
40	$a_{37,9}a_{34,9} \oplus a_{36,9} \oplus a_{37,9} = 0,\ a_{38,13}a_{34,13} \oplus a_{39,13} \oplus a_{38,13} = 0$
41	$a_{39,9}a_{35,9} \oplus a_{40,9} \oplus a_{39,9} = 0,\ a_{40,13}a_{35,13} \oplus a_{35,13} \oplus a_{39,13} = 0$
42	$a_{41,9}a_{36,9} \oplus a_{36,9} \oplus a_{40,9} = 0,\ a_{36,13} = 1$
43	$a_{42,6} = 0,\ a_{37,9} = 0,\ a_{39,6}a_{37,6} \oplus a_{40,6} = 0,$ $a_{42,13}a_{39,13} \oplus a_{41,13}a_{40,13} \oplus a_{39,13} \oplus a_{38,13} \oplus a_{36,13} = 0$
44	$a_{41,6}a_{38,6} \oplus a_{40,6} \oplus a_{41,6} = 0,\ a_{38,13} = 1,\ a_{43,9}a_{40,9} \oplus a_{42,9}a_{41,9} \oplus a_{40,9} \oplus a_{39,9} \oplus a_{37,9} = 0$
45	$a_{43,6}a_{39,6} \oplus a_{44,6} \oplus a_{43,6} = 0,\ a_{39,9} = 0,$ $a_{44,9}a_{41,9}a_{39,9} \oplus a_{39,9}a_{43,9}a_{42,9} \oplus a_{41,9}a_{43,9} \oplus a_{39,9}a_{40,9} \oplus a_{44,9}a_{41,9} \oplus a_{43,9}a_{42,9} = 1$
46	$a_{45,6}a_{40,6} \oplus a_{40,6} \oplus a_{44,6} = 0$
47	$a_{46,30} = 0,\ a_{43,30}a_{41,30} \oplus a_{44,30} = 0,\ a_{41,6} = 0$
48	$a_{45,30}a_{42,30} \oplus a_{44,30} \oplus a_{45,30} = 0,\ a_{47,6}a_{44,6} \oplus a_{46,6}a_{45,6} \oplus a_{44,6} \oplus a_{43,6} \oplus a_{41,6} = 0$
49	$a_{47,30}a_{43,30} \oplus a_{48,30} \oplus a_{47,30} = 0,\ a_{43,6} = 1$
50	$a_{49,30}a_{44,30} \oplus a_{44,30} \oplus a_{48,30} = 0$
51	$a_{50,27} = 1,\ a_{50,28} = 0,\ a_{47,27}a_{45,27} \oplus a_{48,27} = 0,\ a_{47,28}a_{45,28} \oplus a_{48,28} = 0,\ a_{45,30} = 0$
52	$a_{49,27}a_{46,27} \oplus a_{48,27} \oplus a_{49,27} = 0,\ a_{49,28}a_{46,28} \oplus a_{48,28} \oplus a_{49,28} = 0,$ $a_{51,30}a_{48,30} \oplus a_{50,30}a_{49,30} \oplus a_{48,30} \oplus a_{47,30} \oplus a_{45,30} = 0$
53	$a_{51,27}a_{47,27} \oplus a_{52,27} \oplus a_{51,27} = 0,\ a_{51,28}a_{47,28} \oplus a_{52,28} \oplus a_{51,28} = 0,\ a_{47,30} = 0$
54	$a_{53,23} = 0,\ a_{50,23}a_{48,23} \oplus a_{51,23} = 0,\ a_{53,27}a_{48,27} \oplus a_{48,27} \oplus a_{52,27} = 0,$ $a_{53,28}a_{48,28} \oplus a_{48,28} \oplus a_{52,28} = 0$
55	$a_{54,19} = 0,\ a_{51,19}a_{49,19} \oplus a_{52,19} = 0,\ a_{49,23} \oplus a_{51,23} = 0,\ a_{49,27} = 0,\ a_{49,28} = 1$
56	$a_{55,21} = 0,\ a_{52,21} = 1,\ a_{50,21} = 1,\ \oplus a_{53,21} = 1,\ a_{53,19}a_{50,19} \oplus a_{52,19} \oplus a_{53,19} = 0,$ $a_{50,23} \oplus a_{55,23} = 1,\ a_{55,27}a_{52,27} \oplus a_{54,27}a_{53,27} \oplus a_{52,27} \oplus a_{51,27} \oplus a_{49,27} = 0,$ $a_{55,28}a_{52,28} \oplus a_{54,28}a_{53,28} \oplus a_{52,28} \oplus a_{51,28} \oplus a_{49,28} = 0,\ a_{52,21}a_{51,21} \oplus a_{51,21} = 0$
57	$a_{56,14} = 1,\ a_{56,15} = 0,\ a_{54,21}a_{51,21} \oplus a_{53,21} \oplus a_{54,21} = 0,\ a_{56,23}a_{51,23} \oplus a_{51,23} \oplus a_{55,23} = 0,$ $a_{55,19}a_{51,19} \oplus a_{56,19} \oplus a_{55,19} = 0,\ a_{51,27} = 1,\ a_{51,28} = 1$
58	$a_{55,14}a_{52,14} \oplus a_{55,14} \oplus a_{54,14} = 0,\ a_{55,15}a_{52,15} \oplus a_{55,15} \oplus a_{54,15} = 0,$ $a_{56,21}a_{52,21} \oplus a_{57,21} \oplus a_{56,21} = 0,\ a_{57,19}a_{52,19} \oplus a_{52,19} \oplus a_{56,19} = 0,$ $a_{57,23} \oplus a_{56,23} \oplus a_{57,23}a_{55,23} = 0,\ a_{52,23} = 1$
59	$a_{57,14}a_{53,14} \oplus a_{58,14} \oplus a_{57,14} = 0, a_{57,15}a_{53,15} \oplus a_{58,15} \oplus a_{57,15} = 0,\ a_{53,19} = 0,$ $a_{58,21}a_{53,21} \oplus a_{53,21} \oplus a_{57,21} = 0,\ a_{58,23}a_{55,23} \oplus a_{57,23}a_{56,23} \oplus a_{51,23} = 1$
60	$a_{59,14}a_{54,14} \oplus a_{54,14} = 0,\ a_{59,15}a_{54,15} \oplus a_{54,15} \oplus a_{58,15} = 0,$ $a_{59,19}a_{56,19} \oplus a_{58,19}a_{57,19} \oplus a_{56,19} \oplus a_{55,19} \oplus a_{53,19} = 0,\ a_{54,21} = 0,\ a_{54,23} = 1$
61	$a_{55,14} = 0,\ a_{55,15} = 0,\ a_{55,19} = 0,\ a_{60,21}a_{57,21} \oplus a_{59,21}a_{58,21} \oplus a_{57,21} \oplus a_{56,21} \oplus a_{53,21} = 0$ $a_{59,19}a_{57,19} \oplus a_{60,19}a_{58,19} \oplus a_{59,19}a_{58,19} = 1$
62	$a_{61,14}a_{58,14} \oplus a_{60,14}a_{59,14} \oplus a_{58,14} \oplus a_{57,14} \oplus a_{61,14} = 0,\ a_{60,15}a_{59,15} \oplus a_{58,15} = 1$ $a_{61,15}a_{58,15} \oplus a_{60,15}a_{59,15} \oplus a_{58,15} \oplus a_{57,15} \oplus a_{61,15} = 1,\ a_{56,21} = 1$
63	$a_{57,14} = 1,\ a_{57,15} = 1$
64	$a_{60,10} = 0,\ a_{63,10} = 0,\ a_{61,10}a_{58,10} \oplus a_{62,10}a_{59,10} = 1$
67	$a_{62,10}a_{61,10} \oplus a_{60,10} \oplus a_{66,10} = 0$
68	$a_{64,10}a_{62,10} \oplus a_{66,10} = 0$
69	$a_{65,10}a_{64,10} \oplus a_{66,10} = 0$
70	$a_{66,10} = 0$

Table 6. Simplified sufficient conditions of the second preimage differential path of 3-pass HAVAL

S	Sufficient conditions
	$a_{16,31} = 0,\ a_{18,31} = 0,\ a_{20,31} = a_{21,31}, a_{22,31} = 0$
23	$a_{23,31} = 0$
24	$a_{24,31} = 1,\ a_{24,20} = 0$
26	$a_{26,31} = 0$
28	$a_{28,31} = 0, a_{28,20} = 0$
29	$a_{29,20} = 0$
30	$a_{30,20} = 0$
31	$a_{31,20} = 0$
32	$a_{32,20} = 0$
36	$a_{36,13} = 1,\ a_{36,9} = 0,\ a_{34,20}a_{33,20} \oplus a_{35,20} = 0$
37	$a_{37,9} = 0,\ a_{37,6} = 0,\ a_{37,13} = 0,\ a_{33,20}a_{35,20} \oplus a_{36,20}a_{34,20} \oplus a_{35,20}a_{34,20} = 0$
38	$a_{38,13} = 1,\ a_{38,9} = 0,\ a_{34,13}a_{32,13} \oplus a_{35,13} = 0,$
39	$a_{39,6} = 1,\ a_{39,9} = 0,\ a_{35,9}a_{33,9} = 0,\ a_{33,13} \neq a_{35,13}$
40	$a_{40,9} = 0,\ a_{40,6} = 0,\ a_{34,13} \neq a_{39,13}$
41	$a_{41,6} = 0,\ a_{40,13}a_{35,13} \oplus a_{35,13} \oplus a_{39,13} = 0$
42	$a_{42,6} = 0$
43	$a_{43,6} = 1,\ a_{42,13}a_{39,13} \oplus a_{41,13}a_{40,13} \oplus a_{39,13} = 0$
44	$a_{44,30} = 0,\ a_{44,6} = 0,\ a_{42,9}a_{41,9} = 0$
45	$a_{45,6} = 1,\ a_{45,30} = 0,\ a_{41,9}a_{43,9} \oplus a_{44,9}a_{41,9} \oplus a_{43,9}a_{42,9} = 1$
46	$a_{46,6} = 1,\ a_{46,30} = 0$
47	$a_{47,30} = 0,\ a_{43,30}a_{41,30} = 0$
48	$a_{48,30} = 0$
49	$a_{49,27} = 0,\ a_{49,28} = 1$
50	$a_{50,21} = 1,\ a_{50,27} = 1,\ a_{50,28} = 0$
51	$a_{51,27} = 1,\ a_{51,28} = 1,\ a_{51,15} = 0,\ a_{51,14} = 0,$ $a_{47,27}a_{45,27} \oplus a_{48,27} = 0,\ a_{47,28}a_{45,28} \oplus a_{48,28} = 0$
52	$a_{52,19} = 0,\ a_{52,21} = 1, a_{50,30}a_{49,30} = 0,\ a_{52,23} = 1,\ a_{48,27} = 0,\ a_{46,28} = a_{48,28}$
53	$a_{53,19} = 0,\ a_{53,14} = 0,\ a_{53,15} = 1,\ a_{53,21} = 0,\ a_{53,23} = 1,\ a_{47,28} \oplus a_{52,28} = 1$
54	$a_{54,14} = 0,\ a_{54,15} = 0,\ a_{54,19} = 0,\ a_{50,23}a_{48,23} \oplus a_{51,23} = 0,\ a_{52,27} = 0,$ $a_{54,21} = 0,\ a_{54,23} = 1,\ a_{53,28}a_{48,28} \oplus a_{48,28} \oplus a_{52,28} = 0$
55	$a_{55,14} = 0,\ a_{55,15} = 0,\ a_{55,19} = 0,\ a_{55,21} = 0,\ a_{51,19}a_{49,19} = 0,\ a_{49,23} = a_{51,23}$
56	$a_{56,19} = 0,\ a_{56,15} = 0,\ a_{56,14} = 1,\ a_{56,21} = 1,\ a_{50,23} = a_{55,23}$ $a_{54,27}a_{53,27} \oplus a_{55,27} = 1,\ a_{55,28}a_{52,28} \oplus a_{54,28}a_{53,28} \oplus a_{52,28} \oplus a_{51,28} \oplus a_{55,28} = 0$
57	$a_{57,14} = 1,\ a_{57,15} = 1,\ a_{57,21} = 0,\ a_{56,23}a_{51,23} \oplus a_{51,23} \oplus a_{55,23} = 0$
58	$a_{58,21} = 1,\ a_{58,14} = 0, a_{58,15} = 0,\ a_{57,23} \oplus a_{56,23} \oplus a_{57,23}a_{55,23} = 0$
59	$a_{59,21} = 1,\ a_{58,23}a_{55,23} \oplus a_{57,23}a_{56,23} \oplus a_{55,23} \oplus a_{51,23} = 1,\ a_{59,15} = 1$
60	$a_{58,19}a_{57,19} \oplus a_{55,19} = 0,\ a_{60,10} = 0,\ a_{60,15} = 1$
61	$a_{59,19}a_{57,19} \oplus a_{60,19}a_{58,19} \oplus a_{59,19}a_{58,19} = 1,\ a_{61,15} = 1$
62	$a_{60,14}a_{59,14} \oplus a_{61,14} = 1$
63	$a_{63,10} = 0$
64	$a_{61,10}a_{58,10} \oplus a_{62,10}a_{59,10} = 1$
66	$a_{66,10} = 0$
67	$a_{62,10}a_{61,10} = 0$
68	$a_{64,10}a_{62,10} = 0$
69	$a_{65,10}a_{64,10} = 0$

Table 7. Property of the Boolean function f_1

Assumption		Conditions for satisfying the Assumption
$\mathbf{a_{i-1}}[\mathbf{j}]$	$t_i[]$	$a_{i-7} = 0$
	$t_i[j]$	$a_{i-7} = 1, a_{i-3}a_{i-4} \oplus a_{i-2}a_{i-6} \oplus a_{i-5}a_{i-3} \oplus a_{i-5} = 0$
$\mathbf{a_{i-2}}[\mathbf{j}]$	$t_i[]$	$a_{i-6} = 0$
	$t_i[j]$	$a_{i-6} = 1,\ a_{i-3}a_{i-4} \oplus a_{i-7}a_{i-1} \oplus a_{i-5}a_{i-3} \oplus a_{i-5} = 0$
$\mathbf{a_{i-3}}[\mathbf{j}]$	$t_i[]$	$a_{i-4} = a_{i-5}$
	$t_i[j]$	$a_{i-4} \neq a_{i-5},\ a_{i-1}a_{i-7} \oplus a_{i-6}a_{i-2} \oplus a_{i-5} = 0$
$\mathbf{a_{i-4}}[\mathbf{j}]$	$t_i[]$	$a_{i-3} = 0$
	$t_i[j]$	$a_{i-3} = 1,\ a_{i-1}a_{i-7} \oplus a_{i-6}a_{i-2} \oplus a_{i-5}a_{i-3} \oplus a_{i-5} = 0$
$\mathbf{a_{i-5}}[\mathbf{j}]$	$t_i[]$	$a_{i-3} = 1$
	$t_i[j]$	$a_{i-3}a_{i-4} \oplus a_{i-3} = 0,\ a_{i-1}a_{i-7} \oplus a_{i-6}a_{i-2} = 0$
$\mathbf{a_{i-6}}[\mathbf{j}]$	$t_i[]$	$a_{i-2} = 0$
	$t_i[j]$	$a_{i-2} = 1,\ a_{i-3}a_{i-4} \oplus a_{i-7}a_{i-1} \oplus a_{i-5}a_{i-3} \oplus a_{i-5} = 0$
$\mathbf{a_{i-7}}[\mathbf{j}]$	$t_i[]$	$a_{i-1} = 0$
	$t_i[j]$	$a_{i-1} = 1,\ a_{i-3}a_{i-4} \oplus a_{i-6}a_{i-2} \oplus a_{i-5}a_{i-3} \oplus a_{i-5} = 0$

Table 8. Property of the Boolean function f_2

Assumption		Conditions for satisfying the Assumption
$\mathbf{a_{i-1}}[\mathbf{j}]$	$t_i[]$	$a_{i-4}a_{i-6} \oplus a_{i-3} = 0$
	$t_i[j]$	$a_{i-4}a_{i-6} \oplus a_{i-3} = 1,$ $a_{i-6}a_{i-2}a_{i-3} \oplus a_{i-4}a_{i-6} \oplus a_{i-4}a_{i-2}$ $\oplus a_{i-6}a_{i-5} \oplus a_{i-2}a_{i-3} \oplus a_{i-7}a_{i-6} \oplus a_{i-7} = 0$
$\mathbf{a_{i-2}}[\mathbf{j}]$	$t_i[]$	$a_{i-3}a_{i-6} \oplus a_{i-4} \oplus a_{i-3} = 0$
	$t_i[j]$	$a_{i-3}a_{i-6} \oplus a_{i-4} \oplus a_{i-3} = 1,$ $a_{i-4}a_{i-6}a_{i-1} \oplus a_{i-4}a_{i-6} \oplus a_{i-6}a_{i-5} \oplus a_{i-1}a_{i-3} \oplus a_{i-7}a_{i-6} \oplus a_{i-7} = 0$
$\mathbf{a_{i-3}}[\mathbf{j}]$	$t_i[]$	$a_{i-2}a_{i-6} \oplus a_{i-1} \oplus a_{i-2} = 0$
	$t_i[j]$	$a_{i-2}a_{i-6} \oplus a_{i-1} \oplus a_{i-2} = 1,$ $a_{i-4}a_{i-6}a_{i-1} \oplus a_{i-4}a_{i-6} \oplus a_{i-6}a_{i-5} \oplus a_{i-7}a_{i-6} \oplus a_{i-7} = 0$
$\mathbf{a_{i-4}}[\mathbf{j}]$	$t_i[]$	$a_{i-1}a_{i-6} \oplus a_{i-6} \oplus a_{i-2} = 0$
	$t_i[j]$	$a_{i-1}a_{i-6} \oplus a_{i-6} \oplus a_{i-2} = 1$ $a_{i-6}a_{i-2}a_{i-3} \oplus a_{i-6}a_{i-5} \oplus a_{i-1}a_{i-3} \oplus a_{i-2}a_{i-3} \oplus a_{i-7}a_{i-6} \oplus a_{i-7} = 0$
$\mathbf{a_{i-5}}[\mathbf{j}]$	$t_i[]$	$a_{i-6} = 0$
	$t_i[j]$	$a_{i-6} = 1,$ $a_{i-4}a_{i-6}a_{i-1} \oplus a_{i-6}a_{i-2}a_{i-3} \oplus a_{i-4}a_{i-6}$ $\oplus a_{i-4}a_{i-2} \oplus a_{i-1}a_{i-3} \oplus a_{i-2}a_{i-3} \oplus a_{i-7}a_{i-6} \oplus a_{i-7} = 0$
$\mathbf{a_{i-6}}[\mathbf{j}]$	$t_i[]$	$a_{i-1}a_{i-4} \oplus a_{i-2}a_{i-3} \oplus a_{i-4} \oplus a_{i-5} \oplus a_{i-7} = 0$
	$t_i[j]$	$a_{i-1}a_{i-4} \oplus a_{i-2}a_{i-3} \oplus a_{i-4} \oplus a_{i-5} \oplus a_{i-7} = 1,$ $a_{i-4}a_{i-2} \oplus a_{i-1}a_{i-3} \oplus a_{i-2}a_{i-3} \oplus a_{i-7} = 0$
$\mathbf{a_{i-7}}[\mathbf{j}]$	$t_i[]$	$a_{i-6} = 1$
	$t_i[j]$	$a_{i-6} = 0,$ $a_{i-4}a_{i-5}a_i \oplus a_{i-6}a_{i-2}a_{i-3} \oplus a_{i-4}a_{i-6}$ $\oplus a_{i-4}a_{i-2} \oplus a_{i-6}a_{i-5} \oplus a_{i-1}a_{i-3} \oplus a_{i-2}a_{i-3} = 0$

Table 9. The property of the Boolean function f_3

Assumption		Conditions for satisfying the Assumption
$\mathbf{a_{i-1}}[\mathbf{j}]$	$t_i[]$	$a_{i-4} = 1$
	$t_i[j]$	$a_{i-4} = 0, a_{i-4}a_{i-5}a_{i-6} \oplus a_{i-6}a_{i-3} \oplus a_{i-5}a_{i-2} \oplus a_{i-4}a_{i-7} = 0$
$\mathbf{a_{i-2}}[\mathbf{j}]$	$t_i[]$	$a_{i-5} = 0$
	$t_i[j]$	$a_{i-5} = 1,\ a_{i-6}a_{i-3} \oplus a_{i-4}a_{i-7} \oplus a_{i-1}a_{i-4} \oplus a_{i-1} = 0$
$\mathbf{a_{i-3}}[\mathbf{j}]$	$t_i[]$	$a_{i-6} = 0$
	$t_i[j]$	$a_{i-6} = 1,\ a_{i-4}a_{i-5}a_{i-6} \oplus a_{i-5}a_{i-2} \oplus a_{i-4}a_{i-7} \oplus a_{i-1}a_{i-4} \oplus a_{i-1} = 0$
$\mathbf{a_{i-4}}[\mathbf{j}]$	$t_i[]$	$a_{i-5}a_{i-6} \oplus a_{i-7} \oplus a_{i-1} = 0$
	$t_i[j]$	$a_{i-5}a_{i-6} \oplus a_{i-7} \oplus a_{i-1} = 1,\ a_{i-6}a_{i-3} \oplus a_{i-5}a_{i-2} \oplus a_{i-1} = 0$
$\mathbf{a_{i-5}}[\mathbf{j}]$	$t_i[]$	$a_{i-4}a_{i-6} \oplus a_{i-2} = 0$
	$t_i[j]$	$a_{i-4}a_{i-6} \oplus a_{i-2} = 1,\ a_{i-6}a_{i-3} \oplus a_{i-4}a_{i-7} \oplus a_{i-1}a_{i-4} \oplus a_{i-1} = 0$
$\mathbf{a_{i-6}}[\mathbf{j}]$	$t_i[]$	$a_{i-4}a_{i-5} \oplus a_{i-3} = 0$
	$t_i[j]$	$a_{i-4}a_{i-5} \oplus a_{i-3} = 1,\ a_{i-5}a_{i-2} \oplus a_{i-4}a_{i-7} \oplus a_{i-1}a_{i-4} \oplus a_{i-1} = 0$
$\mathbf{a_{i-7}}[\mathbf{j}]$	$t_i[]$	$a_{i-4} = 0$
	$t_i[j]$	$a_{i-4} = 1,\ a_{i-4}a_{i-5}a_{i-6} \oplus a_{i-6}a_{i-3} \oplus a_{i-5}a_{i-2} \oplus a_{i-1}a_{i-4} \oplus a_{i-1} = 0$

Cryptanalysis of LASH

Ron Steinfeld[1], Scott Contini[1], Krystian Matusiewicz[1], Josef Pieprzyk[1], Jian Guo[2], San Ling[2], and Huaxiong Wang[1,2]

[1] Advanced Computing – Algorithms and Cryptography,
Department of Computing, Macquarie University
{rons,scontini,kmatus,josef,hwang}@ics.mq.edu.au

[2] Division of Mathematical Sciences,
School of Physical & Mathematical Sciences
Nanyang Technological University
{guojian,lingsan,hxwang}@ntu.edu.sg

Abstract. We show that the LASH-x hash function is vulnerable to attacks that trade time for memory, including collision attacks as fast as $2^{(4x/11)}$ and preimage attacks as fast as $2^{(4x/7)}$. Moreover, we briefly mention heuristic lattice based collision attacks that use small memory but require very long messages that are expected to find collisions much faster than $2^{x/2}$. All of these attacks exploit the designers' choice of an all zero IV.

We then consider whether LASH can be patched simply by changing the IV. In this case, we show that LASH is vulnerable to a $2^{(7x/8)}$ preimage attack. We also show that LASH is trivially not a PRF when any subset of input bytes is used as a secret key. None of our attacks depend upon the particular contents of the LASH matrix – we only assume that the distribution of elements is more or less uniform.

Additionally, we show a generalized birthday attack on the final compression of LASH which requires $O\left(x2^{\frac{x}{2(1+\frac{107}{105})}}\right) \approx O(x2^{x/4})$ time and memory. Our method extends the Wagner algorithm to truncated sums, as is done in the final transform in LASH.

1 Introduction

The LASH hash function [2] is based upon the provable design of Goldreich, Goldwasser, and Halevi (GGH) [6], but changed in an attempt to make it closer to practical. The changes are:

1. Different parameters for the $m \times n$ matrix and the size of its elements to make it more efficient in both software and hardware.
2. The addition of a final transform [7] and a Miyaguchi-Preneel structure [9] in attempt to make it resistant to faster than generic attacks.

The LASH authors note that if one simply takes GGH and embeds it in a Merkle-Damgård structure using parameters that they want to use, then there are faster than generic attacks. More precisely, if the hash output is x bits, then they

K. Nyberg (Ed.): FSE 2008, LNCS 5086, pp. 207–223, 2008.

roughly describe attacks which are of order $2^{x/4}$ if n is larger than approximately m^2, or $2^{(7/24)x}$ otherwise[1]. These attacks require an amount of memory of the same order as the computation time. The authors hope that adding the second changes above prevent faster than generic attacks. The resulting proposals are called LASH-x, for LASH with an x bit output.

Although related to GGH, LASH is *not* a provable design: no security proof has been given for it. Both the changes of parameters from GGH and the addition of the Miyaguchi-Preneel structure and final transform prevent the GGH security proof from being applied.

Our Results. In this paper, we show:

– LASH-x is vulnerable to collision attacks which trade time for memory (Sect. 4). This breaks the LASH-x hash function in as little as $2^{(4/11)x}$ work (i.e. nearly a cube root attack). Using similar techniques, we can find preimages in $2^{(4/7)x}$ operations. These attacks exploit LASH's all zero IV, and thus can be avoided by a simple tweak to the algorithm.

– Even if the IV is changed, the function is still vulnerable to a short message (1 block) preimage attack that runs in time/memory $O(2^{(7/8)x})$ – faster than exhaustive search (Sect. 5). Our attack works for *any* IV.

– LASH is not a PRF (Sect. 3.1) when keyed through any subset of the input bytes. Although the LASH authors, like other designers of heuristic hash functions, only claimed security goals of collision resistance and preimage resistance, such functions are typically used for many other purposes [5] such as HMAC [1] which requires the PRF property.

– LASH's final compression (including final transform) can be attacked in $O\left(x2^{\frac{x}{2(1+\frac{107}{105})}}\right) \approx O(x2^{x/4})$ time and memory. To do this, we adapt Wagner's generalized birthday attack [12] to the case of truncated sums (Sect. 6). As far as we are aware, this is the fastest known attack on the final LASH compression.

We also explored collision attacks for very long messages using lattice reduction techniques; experiments for LASH-160 suggest that such attacks can find collisions for LASH-160 in significantly less than 2^{80} time and with very little memory. Due to lack of space, we could not include these results here – refer to Sect. 6.2 for a brief summary.

Before we begin, we would like to make a remark concerning the use of large memory. Traditionally in cryptanalysis, memory requirements have been mostly ignored in judging the effectiveness of an attack. However, recently some researchers have come to question whether this is fair [3,4,13]. To address this issue in the context of our results, we point out that the design of LASH is motivated by the assumption that GGH is insufficient due to attacks that use large memory and run faster than generic attacks [2]. We are simply showing

[1] The authors actually describe the attacks in terms of m and n. We choose to use x which is more descriptive.

that LASH is also vulnerable to such attacks so the authors did not achieve what motivated them to change GGH. We also remark that a somewhat more efficient cost-based analysis is possible [11], but page limits prevent us from providing the analysis here.

After doing this work, we have learnt that a collision attack on the LASH compression function was sketched at the Second NIST Hash Workshop [8]. The attack applies to a certain class of circulant matrices. However, after discussions with the authors [10], we determined that the four concrete proposals of x equal to 160, 256, 384, and 512 are not in this class (although certain other values of x are). Furthermore, the attack is on the compression function only, and does not seem to extend to the full hash function.

We remark that our zero IV attacks apply also to the FFT hash function[2] [8] if it were to be used in Merkle-Damgård mode with a zero IV, giving collisions/preimages with complexity $O(2^{x/3})/O(2^{x/2})$, even if the internal state is longer than output length x. However, our preimage attack for non zero IV would not apply due to the prime modulus.

2 Description of LASH

2.1 Notation

Let us define $\mathsf{rep}(\cdot) : \mathbb{Z}_{256} \to \mathbb{Z}_{256}^{8}$ as a function that takes a byte and returns a sequence of elements $0, 1 \in \mathbb{Z}_{256}$ corresponding to its binary representation in the order of most significant bit first. For example, $\mathsf{rep}(128) = (1, 0, 0, 0, 0, 0, 0, 0)$. We can generalize this notion to sequences of bytes. The function $\mathsf{Rep}(\cdot) : \mathbb{Z}_{256}^{m} \to \mathbb{Z}_{256}^{8\cdot m}$ is defined as $\mathsf{Rep}(s) = \mathsf{rep}(s_1)||\ldots||\mathsf{rep}(s_m)$, e.g. $\mathsf{Rep}((192, 128)) = (1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, \ 0, 0, 0)$. Moreover, for two sequences of bytes we define $\oplus$ as the usual bitwise XOR of the two bitstrings.

We index elements of vectors and matrices starting from zero.

2.2 The LASH-x Hash Function

The LASH-x hash function maps an input of length less than 2^{2x} bits to an output of x bits. Four concrete proposals were suggested in [2]: $x = 160$, 256, 384, and 512.

The hash is computed by iterating a compression function that maps blocks of $n = 4x$ bits to $m = x/4$ bytes ($2x$ bits). The measure of n in bits and m in bytes is due to the original paper. Always $m = n/16$. Below we describe the compression function, and then the full hash function.

Compression Function of LASH-x. The compression function is of the form $f : \mathbb{Z}_{256}^{2m} \to \mathbb{Z}_{256}^{m}$. It is defined as

$$f(r, s) = (r \oplus s) + H \cdot [\mathsf{Rep}(r)||\mathsf{Rep}(s)]^T \ , \tag{1}$$

[2] The proposal only specified the compression function and not the full hash function.

where $r = (r_0, \ldots, r_{m-1})$ and $s = (s_0, \ldots, s_{m-1})$ are column vectors[3] belonging to $\mathbb{Z}_{256}^m$. The vector r is called the *chaining variable*. The matrix H is a circulant matrix of dimensions $m \times (16m)$ defined as $H_{j,k} = a_{(j-k) \bmod 16m}$, where $a_i = y_i \bmod 2^8$, and y_i is defined as $y_0 = 54321$, $y_{i+1} = y_i^2 + 2 \pmod{2^{31} - 1}$ for $i > 0$. Our attacks do not use any properties of this sequence. In some cases, our analysis will split the matrix H into a left half H_L and a right half H_R (each of size $m \times 8m$), where H_L is multiplied by the bits of $\mathsf{Rep}(r)$ and H_R by the bits of $\mathsf{Rep}(s)$.

A visual diagram of the LASH-160 compression function is given in Figure 1, where t is $f(r, s)$.

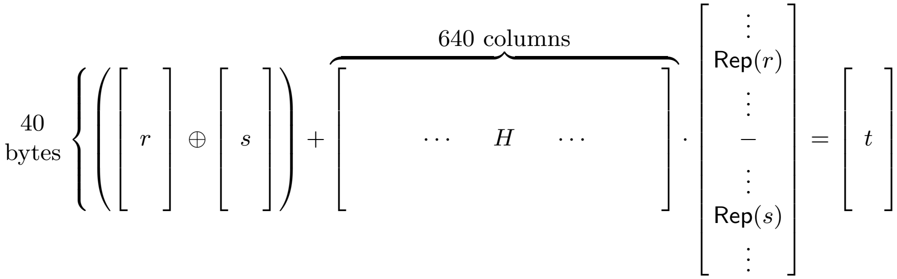


Fig. 1. Visualizing the LASH-160 compression function

The Full Function. Given a message of l bits, padding is first applied by appending a single '1'-bit followed by enough zeros to make the length a multiple of $8m = 2x$. The padded message consists of $\kappa = \lceil (l+1)/8m \rceil$ blocks of m bytes. Then, an extra block b of m bytes is appended that contains the encoded bit-length of the original message, $b_i = \lfloor l/2^{8i} \rfloor \pmod{256}$, $i = 0, \ldots, m-1$.

Next, the blocks $s^{(0)}, s^{(1)}, \ldots, s^{(\kappa)}$ of the padded message are fed to the compression function in an iterative manner as follows: $r^{(0)} := (0, \ldots, 0)$ and then $r^{(j+1)} := f(r^{(j)}, s^{(j)})$ for $j = 0, \ldots, \kappa$. The $r^{(0)}$ is called the IV.

Finally, the last chaining value $r^{(\kappa+1)}$ is sent through a *final transform* which takes only the 4 most significant bits of each byte to form the final hash value h. Precisely, the ith byte of h is $h_i = 16\lfloor r_{2i}/16 \rfloor + \lfloor r_{2i+1}/16 \rfloor$ $(0 \le i < m/2)$.

3 Initial Observations

3.1 LASH is Not a PRF

In some applications (e.g. HMAC) it is required that the compression function (parameterized by its IV) should be a PRF. Below we show that LASH does not satisfy this property by exhibiting a differential that holds with probability 1, independent of the IV.

[3] In this paper, we sometimes abuse notation when there is no confusion in the text: r and s can be both sequences of bytes as well as column vectors.

Assume that r is the secret parameter fixed beforehand and unknown to us. We are presented with a function $g(\cdot)$ which may be $f(r,\cdot)$ or a random function and by querying it we have to decide which one we have.

First we write $H = [H_L || H_R]$ and so (1) can be rewritten as

$$f(r,s) = (r \oplus s) + H_L \cdot \mathsf{Rep}(r)^T + H_R \cdot \mathsf{Rep}(s)^T \ .$$

Setting $s = 0$, we get $f(r,0) = r + H_L \cdot \mathsf{Rep}(r)^T$. Now, for $s' = (128, 0, \ldots, 0)$ we have $\mathsf{Rep}(s') = 100\ldots0$ and so

$$f(r,s') = (r_0 \oplus 128, r_1, \ldots, r_{m-1}) + H_L \cdot \mathsf{Rep}(r)^T + H_R[\cdot, 0] \ ,$$

where $H_R[\cdot,0]$ denotes the first column of the matrix H_R. One can readily compute the difference between $f(r,s')$ and $f(r,0)$:

$$f(r,s') - f(r,0) = H_R[\cdot,0] + (128,0,\ldots,0)^T.$$

Regardless of the value of the secret parameter r, the output difference is a fixed vector equal to $H_R[\cdot,0] + (128,0,\ldots,0)^T$. Thus, using only two queries we can distinguish with probability $1 - 2^{-8m}$ the LASH compression function with secret IV from a randomly chosen function.

The same principle can be used to distinguish LASH even if most of the bytes of s are secret as well. In fact, it is enough for us to control only one byte of the input to be able to use this method and distinguish with probability $1 - 2^{-8}$.

3.2 Absorbing the Feed-Forward Mode

According to [2], the feed-forward operation is motivated by Miyaguchi-Preneel hashing mode and is introduced to thwart some possible attacks on the plain matrix-multiplication construction. In this section we show two conditions under which the feed-forward operation can be described in terms of matrix operations and consequently absorbed into the LASH matrix multiplication step to get a simplified description of the compression function. The first condition requires one of the compression function inputs to be *known*, and the second requires a special subset of input messages.

First Condition: Partially Known Input. Suppose the r portion of the (r,s) input pair to the compression function is known and we wish to express the output $g(s) \stackrel{\text{def}}{=} f(r,s)$ in terms of the unknown input s. We observe that each $(8i+j)$th bit of the feedforward term $r \oplus s$ (for $i = 0,\ldots,m-1$ and $j = 0,\ldots,7$) can be written as

$$\mathsf{Rep}(r \oplus s)_{8i+j} = \mathsf{Rep}(r)_{8i+j} + (-1)^{\mathsf{Rep}(r)_{8i+j}} \cdot \mathsf{Rep}(s)_{8i+j}.$$

Hence the value of the ith byte of $r \oplus s$ is given by

$$\sum_{j=0}^{7} \Big(\mathsf{Rep}(r)_{8i+j} + (-1)^{\mathsf{Rep}(r)_{8i+j}} \cdot \mathsf{Rep}(s)_{8i+j}\Big) \cdot 2^{7-j} =$$

$$\left(\sum_{j=0}^{7} \mathsf{Rep}(r)_{8i+j} \cdot 2^{7-j}\right) + \left(\sum_{j=0}^{7} (-1)^{\mathsf{Rep}(r)_{8i+j}} \cdot \mathsf{Rep}(s)_{8i+j} \cdot 2^{7-j}\right).$$

The first integer in parentheses after the equal sign is just the ith byte of r, whereas the second integer in parentheses is linear in the bits of s with known coefficients, and can be absorbed by appropriate additions to elements of the matrix H_R (defined in Section 2.2). Hence we have an 'affine' representation for $g(s)$:

$$g(s) = (D' + H_R) \cdot \mathsf{Rep}(s)^T + \underbrace{r + H_L \cdot \mathsf{Rep}(r)^T}_{m \times 1 \text{ vector}} , \tag{2}$$

where

$$D' = \begin{bmatrix} J_0 & 0_8 & \dots & 0_8 & 0_8 \\ 0_8 & J_1 & \dots & 0_8 & 0_8 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0_8 & 0_8 & \dots & J_{m-2} & 0_8 \\ 0_8 & 0_8 & \dots & 0_8 & J_{m-1} \end{bmatrix} .$$

Here, for $i = 0, \dots, m-1$, we define the 1×8 vectors $0_8 = [0, 0, 0, 0, 0, 0, 0, 0]$ and

$$J_i = [2^7 \cdot (-1)^{\mathsf{Rep}(r)_{8i}}, 2^6 \cdot (-1)^{\mathsf{Rep}(r)_{8i+1}}, \dots, 2^1 \cdot (-1)^{\mathsf{Rep}(r)_{8i+6}}, 2^0 \cdot (-1)^{\mathsf{Rep}(r)_{8i+7}}] .$$

Second Condition: Special Input Subset. Observe that when bytes of one of the input sequences (say, r) are restricted to values $\{0, 128\}$ only (i.e. only MS bit in each byte can be set), the XOR operation behaves like the byte-wise addition modulo 256. In other words, if $r^* = 128 \cdot r'$ where $r' \in \{0,1\}^m$ then

$$\begin{aligned} f(r^*, s) &= r^* + s + H \cdot [\mathsf{Rep}(r^*)||\mathsf{Rep}(s)]^T \\ &= (D + H) \cdot [\mathsf{Rep}(r^*)||\mathsf{Rep}(s)]^T . \end{aligned} \tag{3}$$

The matrix D recreates values of r^* and s from their representations, similarly to matrix D' above. Then the whole compression function has the linear representation $f(r', s) = H' \cdot [r'||\mathsf{Rep}(s)]^T$ for a matrix H' which is matrix $D + H$ after removing $7m$ columns corresponding to the 7 LS bits of r for each byte. The resulting restricted function compresses $m + 8m$ bits to $8m$ bits using only matrix multiplication without any feed-forward mode.

4 Attacks Exploiting Zero IV

Collision Attack. In the original LASH paper, the authors describe a "hybrid attack" against LASH without the appended message length and final transform. Their idea is to do a Pollard or parallel collision search in such a way that each iteration forces some output bits to a fixed value (such as zero). Thus, the number of possible outputs is reduced from the standard attack. If the total number of possible outputs is S, then a collision is expected after about $\sqrt{S}$ iterations. Using a combination of table lookup and linear algebra, they are able to achieve $S = 2^{(14m/3)}$ in their paper. Thus, the attack is not effective since a collision is expected in about $2^{(7m/3)} = 2^{(7x/12)}$ iterations, which is more than the $2^{x/2}$

iterations one gets from the standard birthday attack on the full LASH function (with the final output transform).

Here, exploiting the zero IV, we describe a similar but simpler attack on the full function which uses table lookup only. Our messages will consist of a number of all-zero blocks followed by one “random” block. Regardless of the number of zero blocks at the beginning, the output of the compression function immediately prior to the length block being processed is determined entirely by the one “random” block. Thus, we will be using table lookup to determine a message length that results in a hash output value which has several bits in certain locations set to some predetermined value(s).

Refer to the visual diagram of the LASH-160 compression function in Fig. 1. Consider the case of the last compression, where the value of r is the output from the previous iteration and the value of s is the message length being fed in. The resulting hash value will consist of the most-significant half-bytes of the bytes of t. Our goal is to quickly determine a value of s so that the most significant half-bytes from the bottom part of t are all approximately zero.

Our messages will be long but not extremely long. Let α be the maximum number of bytes necessary to represent (in binary) any s that we will use. So the bottom $40 - \alpha$ bytes of s are all 0 bytes, and the bottom $320 - 8\alpha$ bits of $\mathsf{Rep}(s)$ are all 0 bits. As before, we divide the matrix H into two halves, H_L and H_R. Without specifying the entire s, we can compute the bottom $40 - \alpha$ bytes of $(r \oplus s) + H_L \cdot \mathsf{Rep}(r)$. Thus, if we precomputed all possibilities for $H_R \cdot \mathsf{Rep}(s)$, then we can use table lookup to determine a value of s that hopefully causes h (to be chosen later) most-significant half-bytes from the bottom part of t to be 0. See the diagram in Fig. 2. The only restriction in doing this is $\alpha + h \leq 40$.

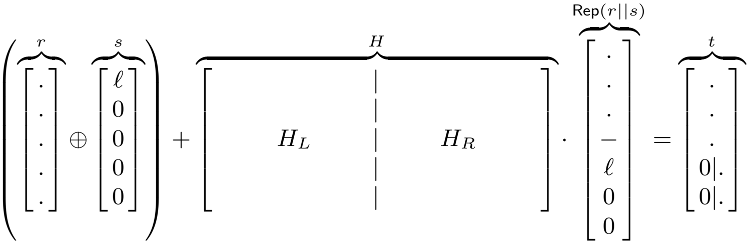


Fig. 2. Visualizing the final block of the attack on the LASH-160 compression function. Diagram is not to scale. Table lookup is done to determine the values at the positions marked with ℓ. Places marked with 0 are set to be zero by the attacker (in the t vector, this is accomplished with the table lookup). Places marked with ‘.’ are outside of the attacker’s control.

We additionally require dealing with the padding byte. To do so, we restrict our messages to lengths congruent to $312 \bmod 320$. Then our “random” block can have anything for the first 39 bytes followed by `0x80` for the 40th byte which is the padding. We then ensure that only those lengths occur in our table lookup by only precomputing $H_R \cdot \mathsf{Rep}(s)$ for values of s of the form $320i + 312$. Thus, we have $\alpha = \lceil \frac{\log 320 + c}{8} \rceil$ assuming we take all values of i less than 2^c. We will aim for $h = c/4$, i.e. setting the bottom $c/4$ half-bytes of t equal to zero. The condition $\alpha + h \leq 40$ is then satisfied as long as $c \leq 104$, which will not be a problem.

Complexity. Pseudocode for the precomputation and table lookup can be found in Table 1 of [11]. With probability $1 - \frac{1}{e} \approx 0.632$, we expect to find a match in our table lookup. Assume that is the case. Due to rounding error, each of the bottom $c/4$ most significant half-bytes of t will either be 0 or -1 (`0xf` in hexadecimal). Thus there are $2^{c/4}$ possibilities for the bottom $c/4$ half-bytes, and the remaining $m - c/4 = x/4 - c/4$ half-bytes ($x - c$ bits) can be anything. So the size of the output space is $S = 2^{x-c+c/4} = 2^{x-3c/4}$. We expect a collision after we have about $2^{x/2-3c/8}$ outputs of this form. Note that with a Pollard or parallel collision search, we will not have outputs of this form a fraction of about $1/e$ of the time. This only means that we have to apply our iteration a fraction of $1/(1 - \frac{1}{e}) \approx 1.582$ times longer, which has negligible impact on the effectiveness of the attack. Therefore, we ignore such constants. Balancing the Pollard search time with the precomputation time, we get an optimal value with $c = (4/11)x$, i.e. a running time of order $2^{(4/11)x}$ LASH-x operations. The lengths of our colliding messages will be order $\leq 2^{c+\log 2x}$ bits. For instance, this attack breaks LASH-160 in as few as 2^{58} operations using 2^{58} storage. The lengths of our colliding messages will be order $\leq 2^{c+\log 2x}$ bits.

Experimental Results. We used this method to find collisions in LASH-160 truncated to the first 12 bytes of the hash: see [11]. The result took one week of cpu time on a 2.4GHz PC with $c = 28$.

Preimage Attack. The same lookup technique can be used for preimage attacks. One simply chooses random inputs and hashes them such that the looked up length sets some of the output hash bits to the target. This involves 2^c precomputation, 2^c storage, and $2^{x-3c/4}$ expected computation time, which balances time/memory to $2^{(4x/7)}$ using the optimal parameter setting $c = 4x/7$.

5 Short Message Preimage Attack on LASH with Arbitrary IV

The attacks in the previous section crucially exploit a particular parameter choice made by the LASH designers, namely the use of an all zero Initial Value (IV) in the Merkle-Damgård construction. Hence, it is tempting to try to 'repair' the LASH design by using a non-zero (or even random) value for the IV. In this section, we show that for any choice of IV, LASH-x is vulnerable to a preimage attack faster than the desired security level of $O(2^x)$. Our preimage attack takes time/memory $O(2^{(7x/8)})$, and produces preimages of short length ($2x$ bits). We give a general description of the attack below with parameter choices for LASH-160 in parentheses. Figure 3, which illustrates the attack for LASH-160, will be particularly useful in understanding our algorithm. For ease of description, we ignore the padding bit, but the reader should be able to see that this can be dealt with as well.

The Attack. Let $f : \mathbb{Z}_{256}^{2m} \to \mathbb{Z}_{256}^{m}$ denote the internal LASH compression function and $f_{out} : \mathbb{Z}_{256}^{2m} \to \mathbb{Z}_{16}^{m}$ denote the final compression function, i.e. the composition of f with the final transform applied to the output of f. Given a target value t_{out} whose LASH preimage is desired, the inversion algorithm finds a single block message $s_{in} \in \mathbb{Z}_{256}^{m}$ hashing to t_{out}, i.e. satisfying

$$f(r_{in}, s_{in}) = r_{out} \qquad \text{(first compression)}$$

and

$$f_{out}(r_{out}, s_{out}) = t_{out} \qquad \text{(final compression)} \ ,$$

where s_{out} is the $8m$-bit (320-bit for LASH-160) binary representation of the length block, and $r_{in} = IV$ is an arbitrary known value. The inversion algorithm proceeds as follows:

Step 1: *Find 2^m (2^{40} for LASH-160) inverses of the final compression.* Using the precomputation-based preimage attack on the final compression function f_{out} described in the previous section (with straightforward modifications to produce the preimage using bits of r_{out} rather than s_{out} and precomputation parameter $c_{out} = (20/7)m$ ($c_{out} \approx 114$ for LASH-160)), compute a list L of 2^m preimage values of r_{out} satisfying $f_{out}(r_{out}, s_{out}) = t_{out}$.

Step 2: *Search for a preimage s_{in} that maps to t_{out}.* Let $c = 3.5m$ ($c = 140$ for LASH-160) be a parameter (later we show that choosing $c = 3.5m$ is optimal). Split the $8m$-bit input s_{in} to be determined into two disjoint parts $s_{in}(1)$ (of length $6m - c$ bit) and $s_{in}(2)$ (of length $2m + c$ bit), i.e. $s_{in} = s_{in}(1)||s_{in}(2)$ (For LASH-160, we have $s_{in}(1)$ of length 100 bits and $s_{in}(2)$ of length 220 bits). We loop through all possibilities for the list L and the set of inputs $s_{in}(1)$ (a total of $2^m \cdot 2^{6m-c} = 2^{7m-c}$ possibilities, or 2^{140} possibilities for LASH-160). For each such possibility, we run the *internal compression function 'hybrid' partial inversion algorithm* described below to compute a matching 'partial preimage' value for $s_{in}(2)$, where by 'partial preimage' we mean that the compression function output $f(r_{in}, s_{in})$ matches target r_{out} on a fixed set of $m + c = 4.5m$ bits (180 bits for LASH-160). We leave the remaining $3.5m$ bits (140 bits for LASH-160) to chance. Thus, for each such computed partial preimage $s_{in} = s_{in}(1)||s_{in}(2)$ and corresponding r_{out} value, we check whether s_{in} is a *full* preimage, i.e. whether $f(r_{in}, s_{in}) = r_{out}$ holds, and if so, output desired preimage s_{in}.

Internal Compression Function 'Hybrid' Partial Inversion Algorithm. For integer parameter c, the internal compression function 'hybrid' partial inversion algorithm is given a $8m$-bit target value t_{in} ($= r_{out}$), an $8m$-bit input r_{in}, and the $(6m - c)$-bit value $s_{in}(1)$, and computes a $(2m + c)$-bit value for $s_{in}(2)$ such that $f(r_{in}, s_{in})$ matches t_{in} on the top $c/7$ bytes as well as on the LS bit of all remaining bytes. Hence, it matches on a total of $m + c$ bits. (For LASH-160, both t_{in} and r_{in} are 320 bits, $s_{in}(1)$ is 100 bits, $s_{in}(2)$ is 220 bits, and $f(r_{in}, s_{in})$

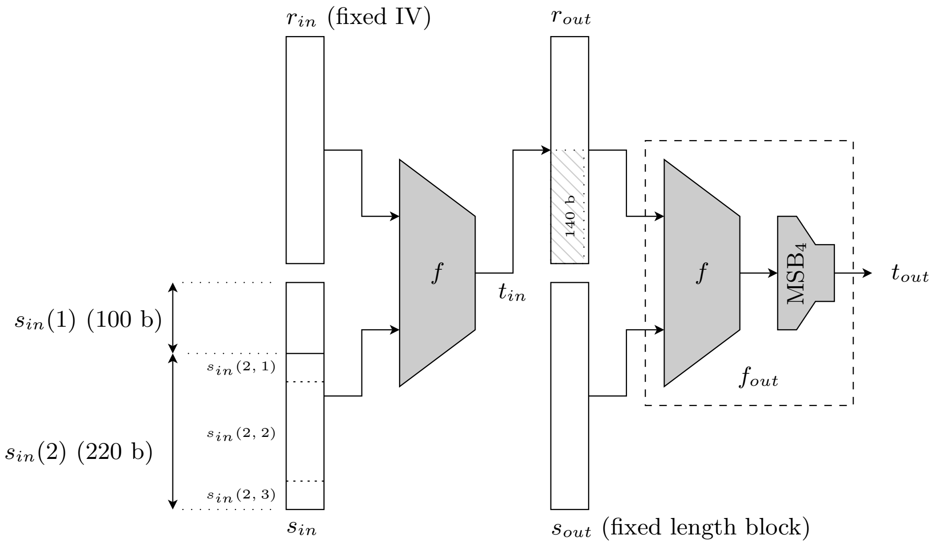


Fig. 3. Illustration of the preimage attack applied to LASH-160

matches t_{in} on all of the bytes in the top half of t_{in} as well as all least significant bits). Some preliminaries are necessary before we explain the algorithm.

From Section 3.2 we know that for known r_{in}, the Miyaguchi-Preneel feedforward term $(r_{in} \oplus s_{in})$ can be absorbed into the matrix by appropriate modifications to the matrix and target vector, i.e. the inversion equation

$$(r_{in} \oplus s_{in}) + H \cdot [\mathsf{Rep}(r_{in})||\mathsf{Rep}(s_{in})]^T = t_{in} \bmod 256, \qquad (4)$$

where H is the LASH matrix, can be reduced to an equivalent linear equation

$$H' \cdot [\mathsf{Rep}(s_{in})]^T = t'_{in} \bmod 256, \qquad (5)$$

for appropriate matrix H' and vector t'_{in} easily computed from the known H, t_{in}, and r_{in}.

We require some notation and a one-time precomputation. We divide $s_{in}(2)$ into 3 parts $s(2,1)$ (length m bits = 40 bits for LASH-160), $s(2,2)$ (length c bits = 140 bits for LASH-160) and $s(2,3)$ (length m bits = 40 bits for LASH-160). For $i = 1, 2, 3$ let $H'(2,i)$ denote the submatrix of matrix H' from (5) consisting of the columns indexed by the bits in $s(2,i)$ (e.g. $H'(2,1)$ consists of the m columns of H' indexed by the m bits of $s(2,1)$). Similarly, let $H'(1)$ denote the submatrix of H' consisting of the columns of H' indexed by the m bits of $s_{in}(1)$.

The one-time precomputation pairs up values of $s(2,2)$ with $s(2,3)$ such that, after multiplying by the corresponding columns of H, the result has 0's in all m least significant bits. To do this, for each of 2^c possible values of $s(2,2)$, we find by linear algebra over $GF(2)$, a matching value for $s(2,3)$ such that

$$[H'(2,2)\ H'(2,3)] \cdot [\mathsf{Rep}(s(2,2))||\mathsf{Rep}(s(2,3))]^T = [0^m]^T \bmod 2. \qquad (6)$$

The entry $s(2,2)||s(2,3)$ is stored in a hash table, indexed by the string of c bits obtained by concatenating 7 MS bits of each of the top $c/7$ bytes of vector y.

We are now ready to describe the internal compression function 'hybrid' partial inversion algorithm, based on [2] (a combination of table lookup and linear algebra), to find $s_{in}(2)$ such that the left and right hand sides of (5) match on the desired $m+c$ bits (180 bits for LASH-160). The gist of the algorithm is to use linear algebra to match the least significant bits and table lookup to match other bits. It works as follows:

- Solving Linear Equations: Compute $s(2,1)$ such that

$$H'(2,1) \cdot [\mathsf{Rep}(s(2,1))]^T = t'_{in} - H'(1) \cdot [\mathsf{Rep}(s_{in}(1))]^T \bmod 2. \qquad (7)$$

  Note that adding (6) and (7) implies that $H' \cdot [\mathsf{Rep}(s_{in}(1))||\mathsf{Rep}(s_{in}(2))]^T = t'_{in} \bmod 2$ with $s_{in}(2) = s(2,1)||s(2,2)||s(2,3)$ for any entry $s(2,2)||s(2,3)$ from the hash table. In other words, all least significant bits will be matched regardless of which entry is taken from the hash table.
- Lookup Hash Table: Find the $s(2,2)||s(2,3)$ entry indexed by the c-bit string obtained by concatenating the 7 MS bits of each of the top $c/7$ bytes of the vector $t'_{in} - H'(2,1) \cdot [\mathsf{Rep}(s(2,1))]^T - H'(1) \cdot [\mathsf{Rep}(s_{in}(1))]^T \bmod 256$. This implies that vector $H' \cdot [\mathsf{Rep}(s_{in}(1))||\mathsf{Rep}(s_{in}(2))]^T$ matches t'_{in} on all top $c/7$ bytes, as well as on the LS bits of all bytes, as required.

Correctness of Attack. For each of 2^m target values r_{out} from list L, and each of the $2^{2.5m}$ possible values for $s_{in}(1)$, the partial preimage inversion algorithm returns $s_{in}(2)$ such that $f(r_{in}, s_{in})$ matches r_{out} on a fixed set of $m+c$ bits. Modelling the remaining bits of $f(r_{in}, s_{in})$ as uniformly random and independent of r_{out}, we conclude that $f(r_{in}, s_{in})$ matches r_{out} on all $8m$ bits with probability $1/2^{8m-(m+c)} = 1/2^{7m-c} = 1/2^{3.5m}$ (using $c = 3.5m$) for each of the $2^{2.5m} \times 2^m = 2^{3.5m}$ runs of the partial inversion algorithm. Assuming that each of these runs is independent, the expected number of runs which produce a full preimage is $2^{3.5m} \times 1/2^{3.5m} = 1$, and hence we expect the algorithm to succeed and return a full preimage.

Complexity. The cost of the attack is dominated by the second step, where we balance the precomputation time/memory $O(2^c)$ of the hybrid partial preimage inversion algorithm with the expected number 2^{7m-c} of runs to get a full preimage. This leads (with the optimum parameter choice $c = 3.5m$) to time/memory cost $O(2^{3.5m}) = O(2^{(7x/8)})$, assuming each table lookup takes constant time. To see that second step dominates the cost, we recall that the first step with precomputation parameter c_{out} uses a precomputation taking time/memory $O(2^{c_{out}})$, and produces a preimage after an expected $O(2^{4m-3c_{out}/4})$ time using $c_{out} + (4m - 3c_{out}/4) = 4m + c_{out}/4$ bits of r_{out}. Hence, repeating this attack 2^m times using m additional bits of r_{out} to produce 2^m distinct preimages is expected to take $O(\max(2^{c_{out}}, 2^{5m-3c_{out}/4}))$ time/memory using $5m + c_{out}/4$ bits of r_{out}. The optimal choice for c_{out} is $c_{out} = (20/7)m \approx 2.89m$, and with this choice the first step takes $O(2^{(20/7)m}) = o(2^{3.5m})$ time/memory and uses $(40/7)m < 8m$ bits of r_{out} (the remaining bits of r_{out} are set to zero).

6 Attacks on the Final Compression Function

This section presents collision attacks on the final compression function f_{out} (including the output transform). For a given $r \in \mathbb{Z}_{256}^m$, the attacks produce $s, s' \in \mathbb{Z}_{256}^m$ with $s \neq s'$ such that $f_{out}(r, s) = f_{out}(r, s')$. To motivate these attacks, we note that they can be converted into a 'very long message' collision attack on the full LASH function, similar to the attack in Sect. 4. The two colliding messages will have the same final non-zero message block, and all preceding message blocks will be zero. To generate such a message pair, the attacker chooses a random $(8m-8)$-bit final message block (common to both messages), pads with a 0x80 byte, and applies the internal compression function f (with zero chaining value) to get a value $r \in \mathbb{Z}_{256}^m$. Then using the collision attack on f_{out} the attacker finds two distinct length fields $s, s' \in \mathbb{Z}_{256}^m$ such that $f_{out}(r, s) = f_{out}(r, s')$. Moreover, s, s' must be congruent to $8m - 8 \pmod{8m}$ due to the padding scheme. For LASH-160, we can force s, s' to be congruent to $8m - 8 \pmod{64}$ by choosing the six LS bits of the length, so this leaves a $1/5^2$ chance that both inputs will be valid.

The lengths s, s' produced by the attacks in this section are very long (longer than $2^{x/2}$). However, we hope the ideas here can be used for future improved attacks.

6.1 Generalized Birthday Attack on the Final Compression

The authors of [2] describe an application of Wagner's generalized birthday attack [12] to compute a collision for the internal compression function f using $O(2^{2x/3})$ time and memory. Although this 'cubic root' complexity is lower than the generic 'square-root' complexity of the birthday attack on the full compression function, it is still higher than the $O(2^{x/2})$ birthday attack complexity on the *full* function, due to the final transformation outputting only half the bytes. Here we describe a variant of Wagner's attack for finding a collision in the *final* compression including the final transform (so the output bit length is x bits). The asymptotic complexity of our attack is $O\left(x2^{\frac{x}{2(1+\frac{107}{105})}}\right)$ time and memory – slightly better than a 'fourth-root' attack. For simplicity, we can call the running time $O(x2^{x/4})$.

The basic idea of our attack is to use the linear representation of f_{out} from Sect. 3.2 and apply a variant of Wagner's attack [12], modified to carefully deal with additive carries in the final transform. As in Wagner's original attack, we build a binary tree of lists with 8 leaves. At the ith level of the tree, we merge pairs of lists by looking for pairs of entries (one from each list) such that their sums have $7 - i$ zero MS bits in selected output bytes, for $i = 0, 1, 2$. This ensures that the list at the root level has 4 zero MS bits on the selected bytes (these 4 MS bits are the output bits), accounting for the effect of carries during the merging process. More precise details are given below.

The attack. The attack uses inputs r, s for which the internal compression function f has a linear representation absorbing the Miyaguchi-Preneel feedforward (see Section 3.2). For such inputs, which may be of length up to $9m$ bit (recall: $m = x/4$), the *final* compression function $f' : \mathbb{Z}_{256}^{9m} \to \mathbb{Z}_{16}^{m}$ has the form

$$f'(r) = MS_4(H' \cdot [\mathsf{Rep}(r)]^T), \tag{8}$$

where $MS_4 : \mathbb{Z}_{256}^{m} \to \mathbb{Z}_{16}^{m}$ keeps only the 4 MS bits of each byte of its input, concatenating the resulting 4 bit strings (note that we use r here to represent the whole input of the linearised compression function f' defined in Section 3.2). Let $Rep(r) = (r[0], r[1], \ldots, r[9m-1]) \in \mathbb{Z}_{256}^{9m}$ with $r[i] \in \{0, 1\}$ for $i = 0, \ldots, 9m-1$. Let $\ell \approx \lfloor \frac{4m}{2(1+107/105)} \rfloor$ (notice that $8\ell < 9m$). We refer to each component $r[i]$ of r as an *input bit*. We choose a subset of 8ℓ input bits from r and partition the subset into 8 substrings $r^i \in \mathbb{Z}_{256}^{\ell}$ $(i = 1, \ldots, 8)$ each containing ℓ input bits, i.e. $r = (r^1, r^2, \ldots, r^8)$. The linearity of (8) gives

$$f'(r) = MS_4(H_1' \cdot [r^1]^T + \cdots + H_8' \cdot [r^8]^T),$$

where, for $i = 1, \ldots, 8$, H_i' denotes the $m \times \ell$ submatrix of H' consisting of the ℓ columns indexed $(i-1) \cdot \ell, (i-1) \cdot \ell + 1, \ldots, i \cdot \ell - 1$ in H'. Following Wagner [12], we build 8 lists $L_1, \ldots, L_8$, where the ith list L_i contains all 2^ℓ possible candidates for the pair (r^i, y^i), where $y^i \overset{\text{def}}{=} H_i' \cdot [r^i]^T$ (note that y^i can be easily computed when needed from r^i and need not be stored). We then use a binary tree algorithm described below to gradually merge these 8 lists into a single list L^3 containing 2^ℓ entries of the form $(r, y = H' \cdot [r]^T)$, where the 4 MS bits in each of the first α bytes of y are zero, for some α, to be defined below. Finally, we search the list L^3 for a pair of entries which match on the values of the 4 MS bits of the last $m - \alpha$ bytes of the y portion of the entries, giving a collision for f' with the output being α zero half-bytes followed by $m - \alpha$ random half-bytes.

The list merging algorithm operates as follows. The algorithm is given the 8 lists $L_1, \ldots, L_8$. Consider a binary tree with $c = 8$ leaf nodes at level 0. For $i = 1, \ldots, 8$, we label the ith leaf node with the list L_i. Then, for each jth internal node n_j^i of the tree at level $i \in \{1, 2, 3\}$, we construct a list L_j^i labelling node n_j^i, which is obtained by merging the lists L_A^{i-1}, L_B^{i-1} at level $i-1$ associated with the two parent nodes of n_j^i. The list L_j^i is constructed so that for $i \in \{1, 2, 3\}$, the entries (r', y') of all lists at level i have the following properties:

- $(r', y') = (r_A' || r_B', y_A' + y_B')$, where (r_A', y_A') is an entry from the left parent list L_A^{i-1} and (r_B', y_B') is an entry from the right parent list L_B^{i-1}.
- If $i \geq 1$, the $\lceil \ell/7 \rceil$ bytes of y' at positions $0, \ldots, \lceil \ell/7 \rceil - 1$ each have their $(7-i)$ MS bits all equal to zero.
- If $i \geq 2$, the $\lceil \ell/6 \rceil$ bytes of y' at positions $\lceil \ell/7 \rceil, \ldots, \lceil \ell/7 \rceil + \lceil \ell/6 \rceil - 1$ each have their $(7-i)$ MS bits all equal to zero.
- If $i = 3$, the $\lceil \ell/5 \rceil$ bytes of y' at positions $\lceil \ell/7 \rceil + \lceil \ell/6 \rceil, \ldots, \lceil \ell/7 \rceil + \lceil \ell/6 \rceil + \lceil \ell/5 \rceil - 1$ each have their $(7-i) = 4$ MS bits all equal to zero.

The above properties guarantee that all entries in the single list at level 3 are of the form $(r, y = H' \cdot [\mathsf{Rep}(r)]^T)$, where the first $\alpha = \lceil \ell/7 \rceil + \lceil \ell/6 \rceil + \lceil \ell/5 \rceil$ bytes of y all have $7 - 3 = 4$ MS bits equal to zero, as required.

To satisfy the above properties, we use a hash table lookup procedure, which aims, when merging two lists at level i, to fix the $7-i$ MS bits of some of the sum bytes to zero. This procedure runs as follows, given two lists L_A^{i-1}, L_B^{i-1} from level $i-1$ to be merged into a single list L^i at level i:

- Store the first component r'_A of all entries (r'_A, y'_A) of L_A^{i-1} in a hash table T_A, indexed by the hash of:
  - If $i = 1$, the 7 MS bits of bytes $0, \ldots, \lceil \ell/7 \rceil - 1$ of y'_A, i.e. string $(MS_7(y'_A[0]), \ldots, MS_7(y'_A[\lceil \ell/7 \rceil - 1]))$.
  - If $i = 2$, the 6 MS bits of bytes $\lceil \ell/7 \rceil, \ldots, \lceil \ell/7 \rceil + \lceil \ell/6 \rceil - 1$ of y'_A, i.e. string $(MS_6(y'_A[\lceil \ell/7 \rceil]), \ldots, MS_6(y'_A[\lceil \ell/7 \rceil + \lceil \ell/6 \rceil - 1]))$.
  - If $i = 3$, the 5 MS bits of bytes $\lceil \ell/7 \rceil + \lceil \ell/6 \rceil, \ldots, \alpha - 1$ of y'_A, i.e. string $(MS_5(y'_A[\lceil \ell/7 \rceil + \lceil \ell/6 \rceil]), \ldots, MS_5(y'_A[\alpha - 1]))$.
- For each entry (r'_B, y'_B) of L_B^{i-1}, look in hash table T_A for matching entry (r'_A, y'_A) of L_A^{i-1} such that:
  - If $i = 1$, the 7 MS bits of corresponding bytes in positions $0, \ldots, \lceil \ell/7 \rceil - 1$ add up to zero modulo $2^7 = 128$, i.e. $MS_7(y'_A[j]) \equiv -MS_7(y'_B[j]) \bmod 2^7$ for $j = 0, \ldots, \lceil \ell/7 \rceil - 1$.
  - If $i = 2$, the 6 MS bits of corresponding bytes in positions $\lceil \ell/7 \rceil, \ldots, \lceil \ell/7 \rceil + \lceil \ell/6 \rceil - 1$ add up to zero modulo $2^6 = 64$, i.e. $MS_6(y'_A[j]) \equiv -MS_6(y'_B[j]) \bmod 2^6$ for $j = \lceil \ell/7 \rceil, \ldots, \lceil \ell/7 \rceil + \lceil \ell/6 \rceil - 1$.
  - If $i = 3$, the 5 MS bits of corresponding bytes in positions $\lceil \ell/7 \rceil + \lceil \ell/6 \rceil, \ldots, \alpha - 1$ add up to zero modulo $2^5 = 32$, i.e. $MS_5(y'_A[j]) \equiv -MS_5(y'_B[j]) \bmod 2^5$ for $j = \lceil \ell/7 \rceil + \lceil \ell/6 \rceil, \ldots, \alpha - 1$.
- For each pair of matching entries $(r'_A, y'_A) \in L_A^{i-1}$ and $(r'_B, y'_B) \in L_B^{i-1}$, add the entry $(r'_A \| r'_B, y'_A + y'_B)$ to list L^i.

Correctness. The correctness of the merging algorithm follows from the following simple fact:

Fact. If $x, y \in \mathbb{Z}_{256}$, and the k MS bits of x and y (each regarded as the binary representation of an integer in $\{0, \ldots, 2^k - 1\}$) add up to zero modulo 2^k, then the $(k-1)$ MS bits of the byte $x + y$ (in $\mathbb{Z}_{256}$) are zero.

Thus, if $i = 1$, the merging lookup procedure ensures, by the Fact above, that the $7 - 1 = 6$ MS bits of bytes $0, \ldots, \lceil \ell/7 \rceil - 1$ of $y'_A + y'_B$ are zero, whereas for $i \geq 2$, we have as an induction hypothesis that the $7 - (i-1)$ MS bits of bytes $0, \ldots, \lceil \ell/7 \rceil - 1$ of both y'_A and y'_B are zero, so again by the Fact above, we conclude that the $7 - i$ MS bits of bytes $0, \ldots, \lceil \ell/7 \rceil - 1$ of $y'_A + y'_B$ are zero, which proves inductively the desired property for bytes $0, \ldots, \lceil \ell/7 \rceil - 1$ for all $i \geq 1$. A similar argument proves the desired property for all bytes in positions $0, \ldots, \alpha - 1$. Consequently, at the end of the merging process at level $i = 3$, we have that all entries (r, y) of list L^3 have the $7 - 3 = 4$ MS bits of bytes $0, \ldots, \alpha - 1$ being zero, as required.

Asymptotic Complexity. The lists at level $i = 0$ have $|L^0| = 2^\ell$ entries. To estimate the expected size $|L^1|$ of the lists at level $i = 1$, we model the entries

(r^0, y^0) of level 0 lists as having uniformly random and independent y^0 components. Hence for any pair of entries $(r^0_A, y^0_A) \in L^0_A$ and $(r^0_B, y^0_B) \in L^0_B$ from lists L^0_A L^0_B to be merged, the probability that the 7 MS bits of bytes $0, \ldots, \lceil \ell/7 \rceil - 1$ of y^0_A and y^0_B are negatives of each other modulo 2^7 is $\frac{1}{2^{\lceil \ell/7 \rceil \times 7}}$. Thus, the total expected number of matching pairs (and hence entries in the merged list L^1) is

$$|L^1| = \frac{|L^0_A| \times |L^0_B|}{2^{\lceil \ell/7 \rceil \times 7}} = \frac{2^{2\ell}}{2^{\lceil \ell/7 \rceil \times 7}} = 2^{\ell + O(1)}.$$

Similarly, for level $i = 2$, we model bytes $\lceil \ell/7 \rceil, \ldots, \lceil \ell/7 \rceil + \lceil \ell/6 \rceil - 1$ as uniformly random and independent bytes, and with the expected sizes $|L^1| = 2^{\ell+O(1)}$ of the lists from level 1, we estimate the expected size $|L^2|$ of the level 2 lists as:

$$|L^2| = \frac{|L^1_A| \times |L^1_B|}{2^{\lceil \ell/6 \rceil \times 6}} = 2^{\ell + O(1)},$$

and a similar argument gives also $|L^3| = 2^{\ell+O(1)}$ for the expected size of the final list. The entries (r, y) of L^3 have zeros in the 4 MS bits of bytes $0, \ldots, \alpha - 1$, and random values in the remaining $m - \alpha$ bytes. The final stage of the attack searches $|L^3|$ for two entries with identical values for the 4 MS bits of each of these remaining $m - \alpha$ bytes. Modelling those bytes as uniformly random and independent we have by a birthday paradox argument that a collision will be found with high constant probability as long as the condition $|L^3| \geq \sqrt{2^{4(m-\alpha)}}$ holds. Using $|L^3| = 2^{\ell+O(1)}$ and recalling that $\alpha = \lceil \ell/7 \rceil + \lceil \ell/6 \rceil + \lceil \ell/5 \rceil = (1/7 + 1/6 + 1/5)\ell + O(1) = \frac{107}{210}\ell + O(1)$, we obtain the attack success requirement

$$\ell \geq \frac{4m}{2(1 + \frac{107}{105})} + O(1) \approx \frac{x}{4} + O(1).$$

Hence, asymptotically, using $\ell \approx \lfloor \frac{x}{2(1+107/105)} \rfloor$, the asymptotic memory complexity of our attack is $O(x 2^{\frac{x}{2(1+\frac{107}{105})}}) \approx O(x2^{x/4})$ bit, and the total running time is also $O(x 2^{\frac{x}{2(1+\frac{107}{105})}}) \approx O(x2^{x/4})$ bit operations. So asymptotically, we have a 'fourth-root' collision finding attack on the final compression function.

Concrete Example. For LASH-160, we expect a complexity in the order of 2^{40}. In practice, the $O(1)$ terms increase this a little. Table 1 summarises the requirements at each level of the merging tree for the attack with $\ell = 42$ (note that at level 2 we keep only 2^{41} of the 2^{42} number of expected list entries to reduce memory storage relative to the algorithm described above). It is not difficult to see that the merging tree algorithm can be implemented such that at most 4 lists are kept in memory at any one time. Hence, we may approximate the total attack memory requirement by 4 times the size of the largest list constructed in the attack, i.e. $2^{48.4}$ bytes of memory. The total attack time complexity is approximated by $\sum_{i=0}^{3} |L^i| \approx 2^{43.3}$ evaluations of the linearised LASH compression function f', plus $\sum_{i=0}^{3} 2^{3-i}|L^i| \approx 2^{46}$ hash table lookups. The resulting attack success probability (of finding a collision on the 72 random output bits among the 2^{37} entries of list L^3) is estimated to be about $1 - e^{-0.5 \cdot 2^{37}(2^{37}-1)/2^{160-88}} \approx 0.86$.

Table 1. Concrete Parameters of an attack on final compression function of LASH-160. For each level i, $|L^i|$ denotes the expected number of entries in the lists at level i, 'Forced Bytes' is the number of bytes whose $7-i$ MS bits are forced to zero by the hash table lookup process at this level, 'Zero bits' is four times the total number of output bytes whose 4 MS bits are guaranteed to be zero in list entries at this level, 'Mem/Item' is the memory requirement (in bit) per list item at this level, 'log(Mem)/List' is the base 2 logarithm of the total memory requirement (in bytes) for each list at this level (assuming that our hash table address space is twice the expected number of list items).

Level (i)	$log(\|L^i\|)$	**Forced Bytes**	**Zero bits**	**Mem/Item**, bit	log(Mem)**/List**, Byte
0	42	6	0	42	45.4
1	42	7	24	84	46.4
2	41	9	52	168	46.4
3	37		88	336	43.4

The total number of input bits used to form the collision is $8\ell = 336$ bit, which is less than the number $9m = 360$ bit available with the linear representation for the LASH compression function.

6.2 Heuristic Lattice-Based Attacks on the Final Compression

We investigated the performance of two heuristic lattice-based methods for finding collisions in truncated versions of the final compression function of LASH. The first reduces finding collisions to a lattice Shortest Vector Problem (SVP). The second uses the SVP as a preprocessing stage and applies a cycling attack with a lattice Closest Vector Problem (CVP) solved at each iteration. Due to lack of space, a detailed description of these attacks and the experimental results obtained can be found elsewhere [11]. The lattice-based attacks have the advantage of requiring very little memory. Preliminary experimental results for LASH-160 [11] give a time complexity estimate below 2^{68} for the CVP-based attack, significantly lower than the desired 2^{80}. Using our SVP-based attack, we found a collision in the final LASH-160 compression function truncated to 120 bits, with time complexity below 2^{36} (much less than the expected 2^{60}).

7 Conclusions

The LASH-x hash function was constructed by taking the GGH provable design [6] and duct taping on components that were intended for heuristic hashing. It is thus a combination of several techniques which are individually sound when applied to ideal components. Our work illustrates that this design strategy does not necessarily yield a secure result when applied to concrete components. In summary, we showed that LASH-x does not meet the designers' security goals nor does it meet other security goals that are typically assumed in heuristic hashing [5].

Acknowledgements. This research was supported in part by Australian Research Council (under grants DP0663452, DP0558773 and DP0665035) and by

Singapore Ministry of Education (under Tier 2 grant T206B2204). Ron Steinfeld's research was supported by the Macquarie University Research Fellowship.

References

1. Bellare, M., Canetti, R., Krawczyk, H.: Keying hash functions for message authentication. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 1–15. Springer, Heidelberg (1996)
2. Bentahar, K., Page, D., Saarinen, M.-J.O., Silverman, J.H., Smart, N.: LASH. In: Second Cryptographic Hash Workshop, August 24–25 (2006)
3. Bernstein, D.J.: Circuits for integer factorization: A proposal. Web page, http://cr.yp.to/papers/nfscircuit.pdf
4. Bernstein, D.J.: What output size resists collisions in a xor of independent expansions? In: ECRYPT Hash Workshop (May 2007)
5. Contini, S., Steinfeld, R., Pieprzyk, J., Matusiewicz, K.: A critical look at cryptographic hash function literature. In: ECRYPT Hash Workshop (May 2007)
6. Goldreich, O., Goldwasser, S., Halevi, S.: Collision-free hashing from lattice problems. Electronic Colloquium on Computational Complexity (ECCC) 3(042) (1996)
7. Lucks, S.: Failure-friendly design principle for hash functions. In: Roy, B. (ed.) ASIACRYPT 2005. LNCS, vol. 3788, pp. 474–494. Springer, Heidelberg (2005)
8. Lyubashevsky, V., Micciancio, D., Peikert, C., Rosen, A.: Provably Secure FFT Hashing (+ comments on "probably secure" hash functions). In: Second Cryptographic Hash Workshop, August 24–25 (2006)
9. Menezes, A.J., van Oorschot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography. CRC Press, Boca Raton (1996)
10. Peikert, C.: Private Communication (August 2007)
11. Steinfeld, R., Contini, S., Matusiewicz, K., Pieprzyk, J., Guo, J., Ling, S., Wang, H.: Cryptanalysis of LASH. Cryptology ePrint Archive, Report 2007/430 (2007), http://eprint.iacr.org/
12. Wagner, D.: A generalized birthday problem. In: Yung, M. (ed.) CRYPTO 2002. LNCS, vol. 2442, pp. 288–303. Springer, Heidelberg (2002)
13. Wiener, M.J.: The full cost of cryptanalytic attacks. J. Cryptol. 17(2), 105–124 (2004)

A (Second) Preimage Attack on the GOST Hash Function

Florian Mendel, Norbert Pramstaller, and Christian Rechberger

Institute for Applied Information Processing and Communications (IAIK),
Graz University of Technology, Inffeldgasse 16a, A-8010 Graz, Austria
Florian.Mendel@iaik.TUGraz.at

Abstract. In this article, we analyze the security of the GOST hash function with respect to (second) preimage resistance. The GOST hash function, defined in the Russian standard GOST-R 34.11-94, is an iterated hash function producing a 256-bit hash value. As opposed to most commonly used hash functions such as MD5 and SHA-1, the GOST hash function defines, in addition to the common iterated structure, a checksum computed over all input message blocks. This checksum is then part of the final hash value computation. For this hash function, we show how to construct second preimages and preimages with a complexity of about 2^{225} compression function evaluations and a memory requirement of about 2^{38} bytes.

First, we show how to construct a pseudo-preimage for the compression function of GOST based on its structural properties. Second, this pseudo-preimage attack on the compression function is extended to a (second) preimage attack on the GOST hash function. The extension is possible by combining a multicollision attack and a meet-in-the-middle attack on the checksum.

Keywords: cryptanalysis, hash functions, preimage attack.

1 Introduction

A cryptographic hash function H maps a message M of arbitrary length to a fixed-length hash value h. A cryptographic hash function has to fulfill the following security requirements:

- *Collision resistance:* it is practically infeasible to find two messages M and M^*, with $M^* \neq M$, such that $H(M) = H(M^*)$.
- *Second preimage resistance:* for a given message M, it is practically infeasible to find a second message $M^* \neq M$ such that $H(M) = H(M^*)$.
- *Preimage resistance:* for a given hash value h, it is practically infeasible to find a message M such that $H(M) = h$.

The resistance of a hash function to collision and (second) preimage attacks depends in the first place on the length n of the hash value. Regardless of how a hash function is designed, an adversary will always be able to find preimages or

K. Nyberg (Ed.): FSE 2008, LNCS 5086, pp. 224–234, 2008.

second preimages after trying out about 2^n different messages. Finding collisions requires a much smaller number of trials: about $2^{n/2}$ due to the birthday paradox. If the internal structure of a particular hash function allows collisions or (second) preimages to be found more efficiently than what could be expected based on its hash length, then the function is considered to be broken.

Recent cryptanalytic results on hash functions mainly focus on collision attacks (see for instance [2,3,14,15,16,17]) but only few results with respect to (second) preimages have been published to date (see for instance [7,9]). In this article, we will present a security analysis with respect to (second) preimage resistance for the hash function specified in the Russian national standard GOST-R 34.11-94. This standard has been developed by *GUBS of Federal Agency Government Communication and Information* and *All-Russian Scientific and Research Institute of Standardization*. The standard also specifies amongst others the GOST block cipher and the GOST signature algorithm.

The GOST hash function is an iterated hash function producing a 256-bit hash value. Since the GOST block cipher is a building block of the hash function, it can be considered as a block-cipher-based hash function. While there have been published several cryptanalytic results regarding the block cipher (see for instance [1,6,8,12,13]), we are not aware of any published security analysis of the GOST hash function besides the work of Gauravaram and Kelsey in [4]. They show that the generic attacks on hash functions based on the Damgård-Merkle design principle can be extended to hash functions with linear/modular checksums independent of the underlying compression function.

In this article, we present an analysis of the GOST hash function. To denote the GOST hash function, we will simply write GOST for the remainder of this article. We exploit the internal structure of GOST to construct pseudo-preimages for the compression function of GOST with a complexity of about 2^{192}. Furthermore, we show how this attack on the compression function of GOST can be extended to a (second) preimage attack on the hash function. The attack has a complexity of about 2^{225} instead of 2^{256} what is expected from a 256-bit hash value. Both attacks are structural attacks in the sense that they are independent of the underlying block cipher.

The remainder of this article is structured as follows. In Section 2, we give a short description of GOST. Section 3 presents the pseudo-preimage attack on the compression function. The extension of the attack on the compression function resulting in the preimage attack is discussed in Section 4. Finally, we present conclusions in Section 5.

2 Description of GOST

GOST is an iterated hash function that processes message blocks of 256 bits and produces a 256-bit hash value. If the message length is not a multiple of 256, an unambiguous padding method is applied. For the description of the padding method we refer to [10]. Let $M = M_1 \| M_2 \| \cdots \| M_t$ be a t-block message (after padding). The hash value $h = H(M)$ is computed as follows (see Fig. 1):

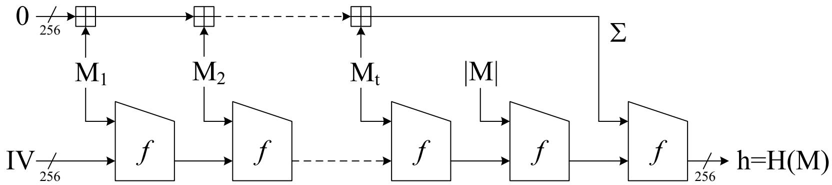


Fig. 1. Structure of the GOST hash function

$$H_0 = IV \tag{1}$$

$$H_i = f(H_{i-1}, M_i) \quad \text{for } 0 < i \leq t \tag{2}$$

$$H_{t+1} = f(H_t, |M|) \tag{3}$$

$$H_{t+2} = f(h_{t+1}, \Sigma) = h \ , \tag{4}$$

where $\Sigma = M_1 \boxplus M_2 \boxplus \cdots \boxplus M_t$, and $\boxplus$ denotes addition modulo 2^{256}. IV is a predefined initial value and $|M|$ represents the bit-length of the entire message prior to padding. As can be seen in (4), GOST specifies a checksum (Σ) consisting of the modular addition of all message blocks, which is then input to the final application of the compression function. Computing this checksum is not part of most commonly used hash functions such as MD5 and SHA-1.

The compression function f of GOST basically consist of three parts (see also Fig. 2): the state update transformation, the key generation, and the output transformation. In the following, we will describe these parts in more detail.

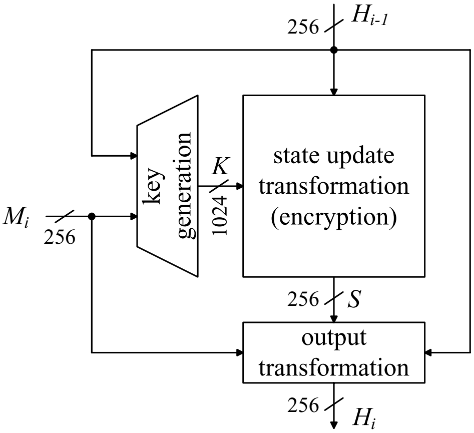


Fig. 2. The compression function of GOST

2.1 State Update Transformation

The state update transformation of GOST consists of 4 parallel instances of the GOST block cipher, denoted by E. The intermediate hash value H_{i-1} is split into four 64-bit words $h_3 \| h_2 \| h_1 \| h_0$. Each 64-bit word is used in one stream of the state update transformation to construct the 256-bit value $S = s_3\|s_2\|s_1\|s_0$ in the following way:

$$
\begin{aligned}
s_0 &= E(k_0, h_0) && (5)\\
s_1 &= E(k_1, h_1) && (6)\\
s_2 &= E(k_2, h_2) && (7)\\
s_3 &= E(k_3, h_3) && (8)
\end{aligned}
$$

where $E(k, p)$ denotes the encryption of the 64-bit plaintext p under the 256-bit key k. We refer to the GOST standard, for a detailed description of the GOST block cipher.

2.2 Key Generation

The key generation of GOST takes as input the intermediate hash value H_{i-1} and the message block M_i to compute a 1024-bit key K. This key is split into four 256-bit keys k_i, *i.e.* $K = k_3\| \cdots \|k_0$, where each key k_i is used in one stream as the key for the GOST block cipher E in the state update transformation. The four keys k_0, k_1, k_2, and k_3 are computed in the following way:

$$
\begin{aligned}
k_0 &= P(H_{i-1} \oplus M_i) && (9)\\
k_1 &= P(A(H_{i-1}) \oplus A^2(M_i)) && (10)\\
k_2 &= P(A^2(H_{i-1}) \oplus \texttt{Const} \oplus A^4(M_i)) && (11)\\
k_3 &= P(A(A^2(H_{i-1}) \oplus \texttt{Const}) \oplus A^6(M_i)) && (12)
\end{aligned}
$$

where A and P are linear transformations and `Const` is a constant. Note that $A^2(x) = A(A(x))$. For the definition of the linear transformation A and P as well as the value of `Const`, we refer to [10], since we do not need them for our analysis.

2.3 Output Transformation

The output transformation of GOST combines the initial value H_{i-1}, the message block M_i, and the output of the state update transformation S to compute the output value H_i of the compression function. It is defined as follows.

$$H_i = \psi^{61}(H_{i-1} \oplus \psi(M_i \oplus \psi^{12}(S))) \tag{13}$$

The linear transformation $\psi : \{0,1\}^{256} \rightarrow \{0,1\}^{256}$ is given by:

$$\psi(\Gamma) = (\gamma_0 \oplus \gamma_1 \oplus \gamma_2 \oplus \gamma_3 \oplus \gamma_{12} \oplus \gamma_{15})\|\gamma_{15}\|\gamma_{14}\| \cdots \|\gamma_1 \tag{14}$$

where Γ is split into sixteen 16-bit words, *i.e.* $\Gamma = \gamma_{15}\|\gamma_{14}\| \cdots \|\gamma_0$.

3 Constructing Pseudo-preimages for the Compression Function of GOST

In this section, we present how to construct a pseudo-preimage for the compression function of GOST. The attack is based on structural weaknesses of the compression function. Since the transformation ψ is linear, (13) can be written as:

$$H_i = \psi^{61}(H_{i-1}) \oplus \psi^{62}(M_i) \oplus \psi^{74}(S) \tag{15}$$

Furthermore, ψ is invertible and hence (15) can be written as:

$$\underbrace{\psi^{-74}(H_i)}_{X} = \underbrace{\psi^{-13}(H_{i-1})}_{Y} \oplus \underbrace{\psi^{-12}(M_i)}_{Z} \oplus S \tag{16}$$

Note that Y depends linearly on H_{i-1} and Z depends linearly on M_i. As opposed to Y and Z, S depends on both H_{i-1} and M_i processed by the block cipher E. For the following discussion, we split the 256-bit words X, Y, Z defined in (16) into 64-bit words:

$$X = x_3 \| x_2 \| x_1 \| x_0 \quad Y = y_3 \| y_2 \| y_1 \| y_0 \quad Z = z_3 \| z_2 \| z_1 \| z_0$$

Now, (16) can be written as:

$$x_0 = y_0 \oplus z_0 \oplus s_0 \tag{17}$$
$$x_1 = y_1 \oplus z_1 \oplus s_1 \tag{18}$$
$$x_2 = y_2 \oplus z_2 \oplus s_2 \tag{19}$$
$$x_3 = y_3 \oplus z_3 \oplus s_3 \tag{20}$$

For a given H_i, we can easily compute the value $X = \psi^{-74}(H_i)$. Now assume, that for the given $X = x_0 \| x_1 \| x_2 \| x_3$, we can find two pairs (H^1_{i-1}, M^1_i) and (H^2_{i-1}, M^2_i), where $H^1_{i-1} \neq H^2_{i-1}$ or $M^1_i \neq M^2_i$, such that both pairs produce the value x_0. Then we know that with a probability of 2^{-192}, these two pairs also lead to the corresponding values x_1, x_2, and x_3. In other words, we have constructed a pseudo-preimage for the given H_i for the compression function of GOST with a probability of 2^{-192}. Therefore, assuming that we can construct 2^{192} pairs (H^j_{i-1}, M^j_i), where $H^j_{i-1} \neq H^k_{i-1}$ or $M^j_i \neq M^k_i$, such that all produce the value x_0, then we have constructed a pseudo-preimage for the compression function.

Based on this short description, we will show now how to construct pseudo-preimages for the compression function of GOST. We will first derive how to construct pairs (H^j_{i-1}, M^j_i), which all produce the same value x_0. This boils down to solving an underdetermined system of equations. Assume, we want to keep the value s_0 in (17) constant. Since $s_0 = E(k_0, h_0)$, we have to find pairs (H^j_{i-1}, M^j_i) such that the values k_0 and h_0 are the same for each pair. We know that h_0 directly depends on H_{i-1}. The key k_0 depends on $H_{i-1} \oplus M_i$. Therefore, we get the following equations:

$$h_0 = a \tag{21}$$
$$m_0 \oplus h_0 = b_0 \tag{22}$$
$$m_1 \oplus h_1 = b_1 \tag{23}$$
$$m_2 \oplus h_2 = b_2 \tag{24}$$
$$m_3 \oplus h_3 = b_3 \tag{25}$$

where a and the b_i's are arbitrary 64-bit values. Note that $k_0 = P(H_{i-1} \oplus M_i) = \bar{B}$, where $\bar{B} = P(B)$ and $B = b_3 \| \cdots \| b_0$, see (9). This is an underdetermined system of equations with $5 \cdot 64$ equations in $8 \cdot 64$ variables over $GF(2)$. Solving this system leads to 2^{192} solutions for which s_0 has the same value. To find pairs (H^j_{i-1}, M^j_i) for which x_0 has the same value, we still have to ensure that also the term $y_0 \oplus z_0$ in (17) has the same value for all pairs. This adds one additional equation (64 equations over $GF(2)$) to our system of equations, namely

$$y_0 \oplus z_0 = c \tag{26}$$

where c is an arbitrary 64-bit value. This equation does not add any new variables, since we know that y_0 depends linearly on $h_3\|h_2\|h_1\|h_0$ and z_0 depends linearly on $m_3\|m_2\|m_1\|m_0$, see (16). To summarize, fixing the value of x_0 boils down to solving an underdetermined equation system with $6 \cdot 64$ equations and $8 \cdot 64$ unknowns over $GF(2)$. This leads to 2^{128} solutions h_i and m_i for $0 \leq i < 4$ and hence 2^{128} pairs (H^j_{i-1}, M^j_i) for which the value x_0 is the same.

Now we can describe how the pseudo-preimage attack on the compression function of GOST works. In the attack, we have to find H_{i-1} and M_i such that $f(H_{i-1}, M_i) = H_i$ for a given value of H_i. The attack can be summarized as follows.

1. Choose random values for b_0, b_1, b_2, b_3 and a. This determines k_0 and h_0
2. Compute $s_0 = E(k_0, h_0)$ and adjust c accordingly such that
$$x_0 = y_0 \oplus z_0 \oplus s_0 = c \oplus s_0$$
holds with $X = \psi^{-74}(H_i)$.
3. Solve the set of $6 \cdot 64$ linear equations over $GF(2)$ to obtain 2^{128} pairs (H^j_{i-1}, M^j_i) for which x_0 is correct.
4. For each pair compute X and check if x_3, x_2 and x_1 are correct. This holds with a probability of 2^{-192}. Thus, after testing all 2^{128} pairs, we will find a correct pair with a probability of $2^{-192} \cdot 2^{128} = 2^{-64}$. Therefore, we have to repeat the attack about 2^{64} times for different choices of b_0, b_1, b_2, b_3 and a to find a pseudo-preimage for the compression function of GOST.

Hence, we can construct a pseudo-preimage for the compression function of GOST with a complexity of about 2^{192} instead of 2^{256} as expected for a compression function with an output size of 256 bits. In the next section, we will show how this attack on the compression function can be extended to a preimage attack on the hash function.

4 A Preimage Attack for the Hash Function

In a preimage attack, we want to find, for a given hash value h, a message M such that $H(M) = h$. As we will show in the following, for GOST we can construct preimages of h with a complexity of about 2^{225} evaluations of the compression function of GOST. Furthermore, the preimage consists of 257 message blocks, *i.e.* $M = M_1 \| \cdots \| M_{257}$. The preimage attack consists basically of four steps as also shown in Figure 3.

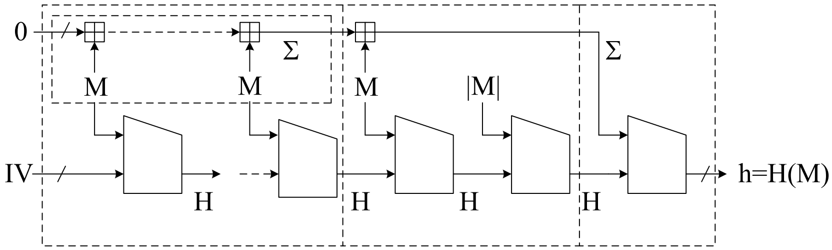


Fig. 3. Preimage Attack on GOST

4.1 STEP 1: Multicollisions for GOST

In [5], Joux introduced multicollisions which can be constructed for any iterated hash function. A multicollision is a set of messages of equal length that all lead to the same hash value. As shown by Joux, constructing a 2^t multicollision, *i.e.* 2^t messages consisting of t message blocks which all lead to the same hash value, can be done with a complexity of about $t \cdot 2^{n/2}$, where n is the bit-size of the hash value. For the preimage attack on GOST, we construct a 2^{256} multicollision. This means, we have 2^{256} messages $M^* = M_1^{j_1} \| M_2^{j_2} \| \cdots \| M_{256}^{j_{256}}$ for $j_1, j_2, \ldots, j_{256} \in \{1, 2\}$ consisting of 256 blocks that all lead to the same hash value H_{256}. This results in a complexity of about $256 \cdot 2^{128} = 2^{136}$ evaluations of the compression function of GOST. Furthermore, the memory requirement is about $2 \cdot 256$ message blocks, *i.e.* we need to store 2^{14} bytes. With these multicollisions, we are able to construct the needed value of Σ^m in STEP 4 of the attack (where the superscript m stands for 'multicollision').

4.2 STEP 2: Pseudo-preimages for the Last Iteration

We construct 2^{32} pseudo-preimages for the last iteration of GOST. For the given h, we proceed as described in Section 3 to construct a list L that consists of 2^{32} pairs (H_{258}, Σ^t) (where the superscript t stands for 'target'). Constructing the list L has a complexity of about $2^{32} \cdot 2^{192} = 2^{224}$ evaluations of the compression function of GOST. The memory requirements in this step come from the storage of 2^{32} pairs (H_{i-1}, M_i), *i.e.* we need to store 2^{32} 512-bit values or 2^{38} bytes.

4.3 STEP 3: Preimages Including the Length Encoding

In this step, we have to find a message block M_{257} such that for the given H_{256} determined in STEP 1, and for $|M|$ determined by our assumption that we want to construct preimages consisting of 257 message blocks, we find a H_{258} that is also contained in the list L constructed in STEP 2. Note that since we want to construct a message that is a multiple of 256 bits, we choose M_{257} to be a full message block and hence no padding is needed. We proceed as follows. Choose an arbitrary message block M_{257} and compute H_{258} as follows:

$$H_{257} = f(H_{256}, M_{257})$$
$$H_{258} = f(H_{257}, |M|)$$

where $|M| = (256 + 1) \cdot 256$. Then we check if the resulting value H_{258} is also in the list L. Since there are 2^{32} entries in L, we will find the right M_{257} with a probability of $2^{-256} \cdot 2^{32} = 2^{-224}$. Hence, after repeating this step of the attack about 2^{224} times, we will find an M_{257} and an according H_{258} that is also contained in the list L. Hence, this step of the attack requires 2^{225} evaluations of the compression function. Once we have found an appropriate M_{257}, also the value Σ^m is determined: $\Sigma^m = \Sigma^t \boxminus M_{257}$.

4.4 STEP 4: Constructing Σ^m

In STEP 1, we constructed a 2^{256} multicollision in the first 256 iterations of the hash function. From this set of messages that all lead to the same H_{256}, we now have to find a message $M^* = M_1^{j_1} \| M_2^{j_2} \| \cdots \| M_{256}^{j_{256}}$ for $j_1, j_2, \ldots, j_{256} \in \{1, 2\}$ that leads to the value of $\Sigma^m = \Sigma^t \boxminus M_{257}$. This can easily done by applying a meet-in-the-middle attack. First, we save all values for $\Sigma_1 = M_1^{j_1} \boxplus M_2^{j_2} \boxplus \cdots \boxplus M_{128}^{j_{128}}$ in the list L. Note that we have in total 2^{128} values in L. Second, we compute $\Sigma_2 = M_{129}^{j_{129}} \boxplus M_{130}^{j_{130}} \boxplus \cdots \boxplus M_{256}^{j_{256}}$ and check if $\Sigma^m \boxminus \Sigma_2$ is in the list L. After testing all 2^{128} values, we expect to find a matching entry in the list L and hence a message $M^* = M_1^{j_1} \| M_2^{j_2} \| \cdots \| M_{256}^{j_{256}}$ that leads to $\Sigma^m = \Sigma^t \boxminus M_{257}$. This step of the attack has a complexity of 2^{128} and a memory requirement of $2^{128} \cdot 2^5 = 2^{133}$ bytes. Once we have found M^*, we found a preimage for GOST consisting of 256+1 message blocks, namely $M^* \| M_{257}$.

The complexity of the preimage attack is determined by the computational effort of STEP 2 and STEP 3, *i.e.* a preimage of h can be found in about $2^{225} + 2^{224} \approx 2^{225}$ evaluations of the compression function. The memory requirements for the preimage attack are determined by finding M^* in STEP 4, since we need to store 2^{133} bytes for the standard meet-in-the-middle attack. Due to the high memory requirements of STEP 4, one could see this part as the bottleneck of the attack. However, the memory requirements of STEP 4 can be significantly reduced by applying a memory-less variant of the meet-in-the-middle attack introduced by Quisquater and Delescaille in [11].

4.5 A Remark on Second Preimages

Note that the presented preimage attack on GOST also implies a second preimage attack. In this case, we are not given only the hash value h but also a message M that results in this hash value. We can construct for any given message a second preimage in the same way as we construct preimages. The difference is, that the second preimage will always consist of at least 257 message blocks. Thus, we can construct a second preimage for any message M (of arbitrary length) with a complexity of about 2^{225} evaluations of the compression function of GOST.

Note that for long messages (more than 2^{32} message blocks) the generic second preimage attack of Gauravaram and Kelsey [4] is more efficient. For instance, a second preimage can be found for a message consisting of about 2^{54} message blocks with a complexity of 2^{203} evaluations of the compression function of GOST and 2^{142} bytes of memory.

5 Conclusion

In this article, we have presented a (second) preimage attack on GOST. Both the preimage and the second preimage attack have a complexity of about 2^{225} evaluations of the compression function and a memory requirement of about 2^{38} bytes. The internal structure of the compression function allows to construct pseudo-preimages with a complexity of about 2^{192}. This alone would not render the hash function insecure but would actually just constitute a certificational weakness. Nevertheless, the fact that we can construct multicollisions for any iterated hash function including GOST and the possibility of applying a meet-in-the-middle attack make the preimage and second preimage attack on GOST possible. More precisely, as opposed to most iterated hash functions, GOST additionally computes a checksum of the single message blocks which is then input to the final application of the compression function. For the preimage attack, we need a certain value in this chain after 256 iterations. The multicollision attack allows to generate a huge set of colliding messages such that we can generate any value for this checksum. Furthermore, a memory-less variant of meet-in-the-middle attack enables us to construct the specific value in an efficient way with respect to both running time and memory requirements.

Acknowledgements

The authors wish to thank Praveen Gauravaram, Mario Lamberger, Vincent Rijmen, and the anonymous referees for useful comments and discussions.

The work in this paper has been supported in part by the Austrian Science Fund (FWF), project P19863 and by the European Commission through the IST Programme under contract IST2002507 932 ECRYPT. The information in this paper is provided as is, and no guarantee or warranty is given or implied that the information is fit for any particular purpose. The user thereof uses the information at its sole risk and liability.

References

1. Biryukov, A., Wagner, D.: Advanced Slide Attacks. In: Preneel, B. (ed.) EUROCRYPT 2000. LNCS, vol. 1807, pp. 589–606. Springer, Heidelberg (2000)
2. Black, J., Cochran, M., Highland, T.: A Study of the MD5 Attacks: Insights and Improvements. In: Robshaw, M.J.B. (ed.) FSE 2006. LNCS, vol. 4047, pp. 262–277. Springer, Heidelberg (2006)
3. Cannière, C.D., Rechberger, C.: Finding SHA-1 Characteristics: General Results and Applications. In: Lai, X., Chen, K. (eds.) ASIACRYPT 2006. LNCS, vol. 4284, pp. 1–20. Springer, Heidelberg (2006)
4. Gauravaram, P., Kelsey, J.: Cryptanalysis of a Class of Cryptographic Hash Functions. Accepted at CT-RSA (2008), http://eprint.iacr.org/2007/277
5. Joux, A.: Multicollisions in Iterated Hash Functions. In: Franklin, M. (ed.) CRYPTO 2004. LNCS, vol. 3152, pp. 306–316. Springer, Heidelberg (2004)

6. Kelsey, J., Schneier, B., Wagner, D.: Key-Schedule Cryptoanalysis of IDEA, G-DES, GOST, SAFER, and Triple-DES. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 237–251. Springer, Heidelberg (1996)
7. Knudsen, L.R., Mathiassen, J.E.: Preimage and Collision Attacks on MD2. In: Gilbert, H., Handschuh, H. (eds.) FSE 2005. LNCS, vol. 3557, pp. 255–267. Springer, Heidelberg (2005)
8. Ko, Y., Hong, S., Lee, W., Lee, S., Kang, J.-S.: Related Key Differential Attacks on 27 Rounds of XTEA and Full-Round GOST. In: Roy, B., Meier, W. (eds.) FSE 2004. LNCS, vol. 3017, pp. 299–316. Springer, Heidelberg (2004)
9. Lamberger, M., Pramstaller, N., Rechberger, C., Rijmen, V.: Second Preimages for SMASH. In: Abe, M. (ed.) CT-RSA 2007. LNCS, vol. 4377, pp. 101–111. Springer, Heidelberg (2006)
10. Michels, M., Naccache, D., Petersen, H.: GOST 34.10 - A brief overview of Russia's DSA. Computers & Security 15(8), 725–732 (1996)
11. Quisquater, J.-J., Delescaille, J.-P.: How Easy is Collision Search. New Results and Applications to DES. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 408–413. Springer, Heidelberg (1990)
12. Saarinen, M.-J.O.: A chosen key attack against the secret S-boxes of GOST (1998), http://citeseer.ist.psu.edu/saarinen98chosen.html
13. Seki, H., Kaneko, T.: Differential Cryptanalysis of Reduced Rounds of GOST. In: Stinson, D.R., Tavares, S. (eds.) SAC 2000. LNCS, vol. 2012, pp. 315–323. Springer, Heidelberg (2001)
14. Wang, X., Lai, X., Feng, D., Chen, H., Yu, X.: Cryptanalysis of the Hash Functions MD4 and RIPEMD. In: Cramer, R.J.F. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 1–18. Springer, Heidelberg (2005)
15. Wang, X., Yin, Y.L., Yu, H.: Finding Collisions in the Full SHA-1. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 17–36. Springer, Heidelberg (2005)
16. Wang, X., Yu, H.: How to Break MD5 and Other Hash Functions. In: Cramer, R.J.F. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 19–35. Springer, Heidelberg (2005)
17. Yu, H., Wang, X., Yun, A., Park, S.: Cryptanalysis of the Full HAVAL with 4 and 5 Passes. In: Robshaw, M.J.B. (ed.) FSE 2006. LNCS, vol. 4047, pp. 89–110. Springer, Heidelberg (2006)

A A Pseudo-collision for the Compression Function

In a similar way as we have constructed a pseudo-preimage in Section 3, we can construct a pseudo-collision for the compression function of GOST. In the attack, we have to find two pairs (H_{i-1}^1, M_i^1) and (H_{i-1}^2, M_i^2), where $H_{i-1}^1 \neq H_{i-1}^2$ or $M_i^1 \neq M_i^2$, such that $f(H_{i-1}^1, M_i^1) = f(H_{i-1}^2, M_i^2)$. The attack can be summarized as follows.

1. Choose random values for a, b_0, b_1, b_2, b_3 and c. This determines x_0.
2. Solve the set of $6 \cdot 64$ linear equations over $GF(2)$ to obtain 2^{128} pairs (H_{i-1}^j, M_i^j) for which x_0 in (17) is equal.
3. For each pair compute $X = x_3 \| x_2 \| x_1 \| x_0$ and save the the triple $(X, H_{i-1}^j,$ $M_i^j)$ in the list L. Note that x_0 is equal for all entries in the list L.

After computing at most 2^{96} candidates for X one expect to find a matching entry (a collision) in L. Note that a collision is likely to exist due to the birthday paradox. Once, we have found a matching entry for X in the list L, we have also found a pseudo-collision for the compression function of GOST, since $H_i = \psi^{74}(X)$, see (16).

Note that memory-less variants of this attack can be devised [11]. Hence, we have a pseudo-collision for the compression function of GOST with a complexity of about 2^{96} instead of 2^{128} as expected for a compression function with an output size of 256 bits.

Guess-and-Determine Algebraic Attack on the Self-Shrinking Generator*

Blandine Debraize[1,2] and Louis Goubin[2]

[1] Gemalto, Meudon, France
blandine.debraize@gemalto.com
[2] Versailles Saint-Quentin-en-Yvelines University, France
Louis.Goubin@prism.uvsq.fr

Abstract. The self-shrinking Generator (SSG) was proposed by Meier and Staffelbach at Eurocrypt'94. Two similar guess-and-determine attacks were independently proposed by Hell-Johansson and Zhang-Feng in 2006, and give the best time/data tradeoff on this cipher so far. These attacks do not depend on the Hamming weight of the feedback polynomial (defining the LFSR in SSG).

In this paper we propose a new attack strategy against SSG, when the Hamming weight is at most 5. For this case we obtain a better tradeoff than all previously known attacks (including Hell-Johansson and Zhang-Feng). Our main idea consists in guessing some information about the internal bitstream of the SSG, and expressing this information by a system of polynomial equations in the still unknown key bits. From a practical point of view, we show that using a SAT solver, such as MiniSAT, is the best way of solving this polynomial system.

Since Meier and Staffelbach original paper, avoiding low Hamming weight feedback polynomials has been a widely believed principle. However this rule did not materialize in previous recent attacks. With the new attacks described in this paper, we show explicitly that this principle remains true.

Keywords: stream cipher, guess-and-determine attacks, multivariate quadratic equations, SAT solver, self-shrinking generator, algebraic cryptanalysis.

1 Introduction

The self-shrinking generator (**SSG**) was proposed by W. Meier and O. Staffelbach at Eurocrypt'94 in [12]. It is a variant of the original Shrinking Generator proposed by Coppersmith, Krawczyk and Mansour in [4,10]. In their paper, they proposed an attack of time complexity $\mathcal{O}(2^{0.75n})$, and $\mathcal{O}(2^{0.69n})$ when the Hamming weight of the feedback polynomial is 3. In [13], Mihaljević proposed a cryptanalysis with minimal time complexity $\mathcal{O}(2^{0.5n})$, with data complexity $\mathcal{O}(n2^{0.5n})$. The amount of keystream is not realistic for large values of the key

* This work has been partially supported by the French Agence Nationale de la Recherche (ANR) under the Odyssee project.

K. Nyberg (Ed.): FSE 2008, LNCS 5086, pp. 235–252, 2008.

size n. An attack on SSG requiring very few keystream data ($2.41n$) is the BBD cryptanalysis proposed in [9] with time complexity $n^{\mathcal{O}(1)}2^{0.656n}$ and equivalent memory complexity. The best tradeoff between time, memory and data complexity today is the Hell and Johansson guess-and-determine cryptanalysis of [8]. A very similar attack has been independently proposed by Zhang and Feng in [14]. For instance the time complexity of this latter attack varies from $\mathcal{O}(2^{0.5n})$ to $\mathcal{O}(n^3 2^{0.666n})$ and data complexity ranges from $\mathcal{O}(n2^{0.5n})$ to $\mathcal{O}(n)$ accordingly. For example with a reasonable amount of keystream of $\mathcal{O}(2^{0.161n})$, it is possible with this attack to recover the key in time $\mathcal{O}(n^3 2^{0.556n})$. The complexity of this attack is independent from the Hamming weight of the feedback polynomial.

In this paper we show that a low Hamming weight for the feedback polynomial defining the LFSR makes the self-shrinking generator even more vulnerable against guess-and-determine attacks. To show this we propose a new type of guess-and-determine attack. We guess some information and then write a system of polynomial equations over GF(2) that we solve by using the SAT solver algorithm MiniSAT. We describe a large family of attacks. Thus as the Hell-Johansson and Zhang-Feng attacks, we can handle with different conditions of attack and data requirements. Our simulations show that for small Hamming weight feedback polynomial, the complexity of our time/data tradeoff is noticeably better.

In Section 2, we briefly describe SAT solvers, the design of the SSG and the principle of our attack. In Section 3 we analyse previous work on this cipher. In Section 4, we describe a special case of our new attack, and in Section 5, we generalize the principle to a family of attacks. Finally in Section 6, we look for the best time/data tradeoff cryptanalysis.

2 Preliminaries

2.1 SAT Solvers

In cryptography the use of SAT solvers to solve polynomial systems over GF(2) has been recently introduced by Bard, Courtois and Jefferson in [1,2]. The method consists of converting the multivariate system into a conjunctive normal form satisfiability (CNF-SAT) problem, and then applying a SAT solver algorithm. It has been used in [3] to cryptanalyse the block cipher Keeloq and in [11] to analyse the reduced version Bivium of the stream cipher Trivium.

The other well-known methods to solve algebraic systems of equations over GF(2) are XL ([5]) and Gröbner bases algorithms like F4 and F5 ([6,7]). Both are linear algebra based methods, their drawback is that they need to store big matrices during the computations and then require a huge amount of memory. Moreover it is unclear how much the sparsity of the initial system helps to reduce the running time of the solving.

SAT solvers behave in a completely different way. Most of them try to find more directly a solution to the system by recursively choosing a variable, first trying to assign it a value and then the other. The important parameters for SAT solvers are the number of clauses, the total length of all the clauses, and the number of variables.

In this paper we use the conversion from algebraic normal form to conjunctive normal form method described in [2], and the SAT solver MiniSAT also proposed in [2]. This conversion method transforms linear equations in long CNF expressions made of long clauses. That is why the method works much better if the linear expressions are short, and, more generally, if the systems are sparse.

2.2 Trade-Off between Guessing and Exploiting Information

In this Section, we specifically consider the case of stream ciphers based on one Linear Feedback Shift Register (LFSR), since the self-shrinking generator belongs to this category. However, the notions defined below can be extended to stream ciphers based on several LFSRs.

Let us suppose the state of the LFSR has length n. At each clock t, the LFSR outputs a bit s_t. The bits $s_0, \cdots, s_{n-1}$ are the bits of the initial state of the LFSR. Here we consider that the initial state of the LFSR is the n-bit key of the cipher.

We call internal sequence at clock t the sequence of bits $S^t = s_0 s_1 ... s_t$. At each clock t, the compression function outputs one bit or an empty word $C(S^t)$. The compression ratio η is the average number of output bits generated by one bit of random internal sequence. For the SSG the compression ratio is $\eta = \frac{1}{4}$.

Definition 1. *The information rate (per bit), which a keystream reveals about the first m bits of the underlying internal bitstream, is denoted by $\alpha(m)$, and defined by $\alpha(m) = \frac{1}{m} I(Z^{(m)}, Y)$, where $Z^{(m)}$ denotes a random $z \in \{0,1\}^m$ and Y a random keystream.*

Then $\alpha(m)$ can be computed as:

$$\alpha(m) = \frac{1}{m} I(Z^{(m)}, Y) = \frac{1}{m}\left(H(Z^{(m)}) - H(Z^{(m)}|Y)\right) = 1 - \frac{1}{m} H(Z^{(m)}|Y)$$

We prove in appendix A that the information rate is constant for the self-shrinking generator and that its value is $\frac{1}{4}$.

For a stream cipher based on one LFSR with a constant information rate and a constant compression ratio, there is always a better attack than exhaustive search, by exploiting the leakage of information given by the keystream. For m keystream bits, this leakage is an amount of $\alpha m/\eta$ bits of information. The entropy of the guess to recover the m/η first internal sequence bits is then $H(Z^{(m)}|Y) = (1-\alpha)\frac{m}{\eta}$. Recovering the n key bits requires then a complexity $\mathcal{O}(2^{(1-\alpha)n})$. This attack has been described in [12]. One way to improve this attack is to decrease the amount of information we guess. In this case we cannot recover directly all the consecutive bits of the initial state of the LFSR, but only part of them. If we guess an amount of information h on the internal sequence per keystream bit, what we obtain is an amount of $h + \alpha/\eta$ per keystream bit. The ratio "guessed information"/"total information known per keystream bit" is then

$$\frac{h}{h + \frac{\alpha}{\eta}}$$

where $\frac{\alpha}{\eta}$ is a constant (here equal to 1). Therefore the smaller h gets, the smaller this ratio becomes. This means that when h decreases, the amount of "guessed information" staying the same, the obtained "total information" increases.

Decreasing the amount of information on the internal sequence we guess per keystream bit seems then to be a good strategy. It is the adopted strategy throughout this paper. The greatest issue is the following: once we have obtained enough information, how to exploit it to recover the key. This will be discussed in detail in this paper for the case of the self-shrinking generator.

2.3 Description of the Self-Shrinking Generator

The self-shrinking generator consists of one LFSR, and a shrinking component that uses a compression function C. Let $K = (K_0, \cdots, K_{n-1})$ be a secret key, and let $s^0 = K$ be the initial state of the LFSR. At each clock $t = 0, 1, 2, \cdots$, the new state s^t is computed as $s^t = L(s^{t-1})$, with L being the multivariate linear transformation corresponding to the connection polynomial of the LFSR. Therefore $s^t = L^t(K_0, \cdots, K_{n-1})$, and every bit s_i^t of the state at time t can be written as a known linear combination of the key bits $K_0, \cdots, K_{n-1}$.

Now we define the compression function. Let f be a function defined as follows:

$$f : \{0,1\}^2 \longrightarrow \{0, 1, \varepsilon\}$$

such that $f(a, b) = b$ if $a = 1$, and $f(a, b) = \varepsilon$ (the empty word) if $a = 0$. This compression function can be extended to compress sequences of bits of arbitrary length as follows. Let $x_0\, x_1 \cdots x_{r-1}$ be a bitstream of length r generated by the LFSR. The output keystream of the SSG generator will be $C(x_0\, x_1 \cdots x_{r-1})$, which is defined as $f(x_0, x_1)\, f(x_2, x_3) \cdots f(x_{r-2}, x_{r-1})$ with the computation being done in the free monoid $\{0,1\}^*$ (which means that we simply concatenate these strings of bits). The resulting compressed sequence $C(x_0\, x_1 \cdots x_{r-1})$ has length at most $\lceil \frac{r}{2} \rceil$. This length is hard to predict and depends on the number of pairs of consecutive bits such that $f(x_i, x_{i+1}) = \varepsilon$ (*i.e.* $x_i = 0$ and no bit is output).

3 Previous Work and Known Attacks

3.1 The Meier and Staffelbach Attack

The attack described in [12] is the attack we refereed to in Section 2.2. It consists of guessing all the consecutive bits of an internal sequence s of length n that are not revealed by the keystream. As the compression ratio is $\frac{1}{4}$, the amount of unknown bits is on average $\frac{3n}{4}$. As announced in Section 2.2, the complexity of this attack is $2^{\frac{3}{4}n}$.

Two completely different attacks were proposed in 2001 [15] of complexity $\mathcal{O}(2^{0.694n})$, and in 2002 [9] of complexity $n^{\mathcal{O}(1)} 2^{0.656n}$, for which we will not go into detail in this paper.

There are two ways of improving Meier and Staffelbach attack. The first one consists of reducing the amount of information we guess, as we describe in

Section 3.2. The second one consists in looking for the best case through the keystream, as we briefly describe in Section 3.3.

3.2 Improvement

It is easy to improve this attack by decreasing the amount of information we guess. The known method we explain here can be found in [8]. Each bit x_i of the pseudo-random sequence corresponds to two consecutive bits 1 and x_i in the internal sequence s. Then it is possible, instead of guessing the values of all the bits of the internal sequence, to guess only the values of the subsequence s' made of the even bits of s $(x_0, x_2, \cdots, x_{2n}, \cdots)$. It is equivalent to guessing the position of the pairs $(1, x_i)$ in s. We show now that this decreases the amount of information we guess per bit. Let us suppose that $x_0 = 1$. The probability for the number of "0" to be k before the next "1" in s' is $\frac{1}{2^{k+1}}$. Consequently the entropy for this information is

$$H(L) = \sum_{j=0}^{+\infty} \frac{j+1}{2^{j+1}} = 2$$

Let us suppose we get a sequence of m bits of keystream. The entropy for guessing the values of the corresponding internal sequence s' (bits in even positions) is then $2m$. Therefore we can guess all these values with an average about 2^{2m} guesses. We have seen that the i-th bit of the keystream is equal to the odd bit following the i-th even "1" of s. Once we get the positions of the "1"s in the internal bitstream, we know the values and positions of $2m + m = 3m$ bits on average. Therefore m must be about $\frac{n}{3}$, assuming there is no redundancy in the information.

How to exploit this information? Here it is very simple, as each internal sequence bit equals a linear expression of the key bits. We have then obtained a system of linear equations. The non-redundancy of the information obtained by our guess is expressed by the consistency of this linear system.

We observe that in this attack the ratio "information guessed"/"information obtained" is $\frac{2}{3}$.

3.3 Mihaljević Attack

This attack is described in [13]. Let us consider again the subsequence s' of the internal bitstream made of the even bits. When we know that $\frac{n}{2}$ consecutive bits of s' are "1"s, we know n consecutive bits of s. The attack consists in looking for this case through the keystream. To each keystream subsequence of $\frac{n}{2}$ bits corresponds an n-bit internal bitstream sequence. If by running the stream cipher on this sequence we do not obtain right values for the keystream, we try on the following $\frac{n}{2}$ bits sequence of keystream, etc.

Of course the drawback of this attack is the huge amount of necessary keystream bits: about $\frac{n}{2} \cdot 2^{\frac{n}{2}}$. This is why [13] describes a family of attacks with time complexity varying from $\mathcal{O}(2^{\frac{n}{2}})$ (this attack) to $\mathcal{O}(2^{\frac{3n}{4}})$ (the attack of Section 3.1), and the required keystream length ranging from $2^{\frac{n}{2}}$ to $2^{\frac{n}{4}}$ accordingly.

The other tradeoff between the attack allowing the best complexity estimation and the attack described at Section 3.2 is studied in [14] and [8]. The attack strategy is the same in both papers, but in [8], an improvement is proposed when the available keystream is very short (less than $2^{0.05n}$). As our final attack will only focus on larger keystream amounts, we will only take into account the common part of [14] and [8] in this paper. We will briefly describe it in Section 6.

4 Principle of Our Attack

Our aim is to generalize the method described at Section 3.2. In this attack we guess some bit values and solve the system of linear equations by a Gaussian elimination when the system of linear equations has rank n.

To adopt a more general point of view on this attack, we can say that we exploit the information we have obtained when its amount is sufficient, *i.e.* when we have obtained n bits of information on the key (recall that the key is the initial state of the LFSR). In Section 3.2 we exploit this information by a linear algebra method. Each linear equation in the key bits represents one bit of information. Here the non-redundancy of the information obtained is guaranteed by the independence of the linear equations.

In the following, we keep this point of view. We guess some information on the internal sequence and directly compute the total amount of information we have obtained. The second step consists then in exploiting this information by completely describing it by a system of polynomial equations and solving this system with algebraic techniques.

4.1 Guessing Information

In the attack of Section 3.1, the amount of information that is guessed per keystream bit is 3. In the attack of Section 3.2, it is 2. What we want to do here is to further decrease this amount of guessed information per bit. Instead of guessing the positions of the “1”s of the subsequence s' made of the even bits of the internal sequence, such as in the attack of Section 3.2, we guess the positions of one such bit out of two.

Let us consider a sequence of keystream bits $x_i, x_{i+1}, \cdots, x_{i+k}, \cdots$. Each of these bits x_j correspond to a pair $(1, x_j)$ in the internal bitstream s. Then we guess the positions of the corresponding pairs for $x_i, x_{i+2}, x_{i+4}, \cdots, x_{i+2k'}, \cdots$. Thus for example the precise position of the pair corresponding to x_{i+1} is unknown but ranges between the position of the pair corresponding to x_i and the position of the pair corresponding to x_{i+2}.

Let us define for this attack a “block” of internal sequence bits: each block contains two pairs beginning by 1 and the pairs beginning by “0” until the next “1” in the sequence. This means that each block begins by a “1”. For example, if the internal sequence is :

$$01\ 10\ 00\ 01\ 10\ 00\ 10\ 00 \cdots,$$

the first block we find for this sequence is 10 00 01 10 00.

To know the position of one 1 out of two in s', it is enough to guess the size of consecutive blocks of s, *i.e.* to guess the number of pairs beginning by 0 in each block. The probability to have k pairs beginning by 0 in a block is the number of ways of distributing k bits among 2 places multiplied by $\frac{1}{2^{k+2}}$. For any q, the number of possibilities to distribute k bits among q places is $\binom{q-1+k}{k}$. The probability is then here $\frac{k+1}{2^{k+2}}$. The entropy of the information guessed by keystream bit is:

$$H = -\frac{1}{2}\sum_{k\geq 0} \frac{\binom{k+1}{k}}{2^{k+2}} \log(\frac{\binom{k+1}{k}}{2^{k+2}}) \approx 1.356$$

This information describes the fact that we know that the first even bit of the block is "1", and that the other even bits are all "0" but one. The total amount of information we know on this block comes from this information and from the fact that we know the values of the keystream bits corresponding to the even "1"s of the block, *i.e.* two bits of information. The average information we know about one block is then $2 \times 1.356 + 2$, and the known information per keystream bit is $1.356 + 1 = 2.356$. Thus for m bits of keystream, we get $2.356m$ bits of information if there is no redundancy. Then m must be approximately $\frac{n}{2.356}$ and the average complexity for the guessing part of the attack is $2^{\frac{1.356n}{2.356}} = 2^{0.575n}$.

4.2 Exploiting the information

The next stage of the attack consists in exploiting the information we have obtained. This information cannot be expressed only by linear GF(2) relations any more. But as we will see now, it is possible to describe it by quadratic equations. To ease the understanding, we call "subblock" all the pairs of a block but the first one. What we have to describe for each block is:

1. The fact that the first and second bits of the block are known. This can still be described by linear relationships.
2. The fact that only one pair among the pairs of the subblock begins by "1". This information can be divided into two parts:
   - There is at most one "1" among the even bits of the subblock. This means that for each even bit of the subblock x_i, if x_j is another even bit of the subblock, we have:

     $$(x_i = 1) \Rightarrow (x_j = 0)$$

     This is equivalent to : $x_i x_j = 0$. Then this part of the information can be described by $\binom{k}{2}$ quadratic equations in the internal sequence bits.
   - There is at least one "1" among the even bits ot the subblock. This is described by a linear equation:

     $$\bigoplus_{j=1}^{k+1} x_{i_j} = 1$$

     where the x_{i_j} are all the even bits of the subblock.

3. The fact that the bit of the pair beginning by "1" in the subblock is known. This is described by the fact that for each even bit x_j of the subblock,

$$(x_j = 1) \Rightarrow (x_{j+1} = e)$$

where e is the corresponding keystream bit. It can be translated by $k+1$ quadratic boolean equations:

$$x_j(x_{j+1} + e) = 0$$

As the composition of linear functions with quadratic equations is still quadratic, those equations can be written as quadratic equations in the key bits. We have then obtained a system of quadratic equations over the field GF(2), completely describing the key.

When the blocks are short, it is possible to find some other equations describing the information. It is interesting to have the most overdefined possible system of equations if programs like Gröbner basis algorithm or XL are used to solve the system. But in this paper we use SAT solver algorithms for which working on very overdefined systems is not the best strategy. That is why we do not add these additional equations in our systems.

We give here the results of our computations on these systems of equations for different sizes of LFSR state n and three different Hamming weights hw for the feedback polynomial of the LFSR:

Table 1. MiniSAT computations on quadratic systems of equations

	$hw = 5$	$hw = 6$	$hw = 7$
$n = 128$	$0.02s$	$0.03s$	$0.05s$
$n = 256$	$0.025s$	$0.046s$	$62s$
$n = 512$	$0.127s$	$> 24h$	$> 24h$
$n = 1024$	$122.25s$	$> 24h$	$> 24h$

5 Generalisation of the Attack

5.1 Guessing Information

This method can be generalized. In Section 4, we have chosen to guess the position of one even "1" in the internal sequence out of $q = 2$. Now we can choose to guess the position of one 1 out of q bits, with $q \geq 2$. This is again equivalent to guessing the length of the "blocks" made of the consecutive bits of the internal sequence containing q pairs of bits beginning by "1" and the other pairs beginning by "0" until the next even "1".

Each such block correspond to q keystream bits. The average entropy per keystream bit to guess the length of consecutive blocks is then:

$$H(q) = -\frac{1}{q} \sum_{k \geq 1} \frac{\binom{q-1+k}{k}}{2^{q+k}} \log(\frac{\binom{q-1+k}{k}}{2^{q+k}})$$

For example, when $q = 3$, $H(q) = 1.0385$.

As explained in Section 2.2, the total amount of information we obtain per keystream bit is $1 + H(q)$. If there is no redundancy, it is then necessary to guess the length of the blocks corresponding to $\frac{n}{1+H(q)}$ keystream bits and the average complexity of the guess is $2^{\frac{H(q)}{1+H(q)}n}$.

Table 2. Average complexity of the guess for various values of q

	$q = 2$	$q = 3$	$q = 4$	$q = 5$
Complexity	$2^{0.575n}$	$2^{0.509n}$	$2^{0.458n}$	$2^{0.417n}$

5.2 Solving the Polynomial System - Computational Results

As in Section 4.2, we need to completely describe the amount of information we have, by means of polynomial GF(2) equations. It is possible to describe it with equations of degree at most q, in a way very similar to the one proposed at Section 4.2. We give details in Appendix B.

Moreover, it is possible to show that for small blocks, the degree of the equations decreases. If Gröbner bases are used, it is well known that the smaller the degree is, the faster the attack is also. With SAT solvers, even if this correlation is not so clear, our computations showed that the complexity gets smaller when the degree of polynomials gets smaller. This tends to show that the shorter the blocks are, the faster the complexity of solving the system is. We will exploit this at Section 6.

We have written the systems of equations for $q = 3$ and $q = 4$ for values of n ranging from 128 to 512. We fixed the value of the Hamming weight of the feedback polynomial to 5 as greater values seem to lead to much slower attacks. We then applied our SAT solver algorithm on these systems. We give the results of the computations in table 3.

Table 3. MiniSAT computations on quadratic systems of equations for q=3 and q=4

	$n = 128$	$n = 256$	$n = 512$
$q = 3$	$2.28s$	$80s$	$2716s$
$q = 4$	$14s$	$1728s$	$> 24h$

6 Improvement of the General Attack

In the previous Sections, we have seen that the basic attack of [12] can be extended in two directions. The first one (first proposed by Mihaljević in [13]) looks for a tradeoff between time complexity and required keystream length. The second one, especially studied in this paper, looks for a tradeoff between the cost of guessing information and the cost of exploiting this information. The best attack consists then of choosing the best tradeoff in both directions at the same time.

In [14] and [8], an attack is proposed that is already a tradeoff between a similar attack as the one described in Section 3.2 and the best time complexity attack proposed in [13], when the length of keystream is maximal. The authors guess all the even bits of a sequence of the internal bitstream of length l, assuming that the rate of "1"s in these even bits is at least α (with a fixed $\alpha > \frac{1}{2}$). They choose the value l, depending on α, in order to have enough information to recover the key by a Gaussian elimination once they have guessed all the even bits of the sequence. In order to find such a sequence, they go through the keystream. The time and data complexity completely depend on the value chosen for α. For instance, the authors of [14] Zhang and Feng obtain a time complexity of $\mathcal{O}(n^3 2^{\frac{n}{1+\alpha}})$.

In Section 5, we denoted by q the number of even "1"s in a block, and considered guessing the position of one even "1" out of q in the internal sequence. In this model, Hell-Johansson and Zhang-Feng attacks correspond to $q = 1$. Our aim in this Section is to find the best tradeoff for $q > 1$.

In order to achieve this, we choose to limit the length of the blocks to a value $k' = 2k$, where $k \geq q$. The probability for a block to have length $2k$ is $\frac{\binom{k-1}{q-1}}{2^k}$, where $\binom{k-1}{q-1}$ is the number of possibilities for the even bits, assuming the first even bit is "1" and there are $q-1$ other "1"s among the even bits of the block. Thus the probability for a block to have length at most $2k$ is:

$$p_{q,k} = \sum_{j=q}^{k} \frac{\binom{j-1}{q-1}}{2^j}$$

If the number of blocks for which we guess the position is l, then the probability for all the blocks to have length at most $2k$ is $(p_{q,k})^l$.

To compute this value l, we need to know the amount of information we have obtained when all the lengths of the blocks are fixed. The entropy leakage provided by the keystream gives q bits of information per block. Then if we call h the amount of information we guess for one block, the total amount of information we then know is $h + q$.

Let us compute $h_{q,k}$, that is h for a block of length $2k$. This information only concerns the even bits of the block. The number of possibilities for the even bits is $\binom{k-1}{q-1}$, *i.e.* the number of manners to distribute the $q-1$ even 1s among the $k-1$ even bits of the subblock made of all the pairs of the block but the first one. This leads to an entropy of $\log(\binom{k-1}{q-1})$. This quantity is the information we still need to guess to have the full knowledge about the even bits of the block, that is k bits of information. Thus the amount we already know (*i.e.* what we have guessed on the even bits) is

$$h_{q,k} = k - \log(\binom{k-1}{q-1})$$

Then for each q we need to find $h_{q,min}$, *i.e.* the minimum of $h_{q,k}$ over all k. We found that for $q = 2$, the minimum of the function holds when $k = 2$, for $q = 3$

when $k = 4$ and for $q = 4$ when $k = 6$. We give the values of these minima in table 4.

Table 4. Minimal information known for the even bits of one block

	$q = 2$	$q = 3$	$q = 4$	$q = 5$
$h_{q,min}$	2	2.415	2.678	2.87

The minimal information we know about one block is $h_{q,min} + q$. We need an information of n bits to recover the key. We still suppose that there is no redundancy in this information. We now can compute the number of blocks l for which we guess the positions, as we know that

$$l(h_{q,min} + q) = n$$

and we obtain $l = \frac{n}{h_{q,min}+q}$.

Our attack is described in algorithm 6.1.

Algorithm 6.1. Our Attack

```
INPUT : $q$,$k$, and a sequence of keystream of length $N$
OUTPUT: values of the $n$ key bits
PROCESSING:
compute $l$ depending on $q$ and $k$
For all the $k^l$ possibilities for the length of the $l$ blocks:
        For $j = 0$ to $N - kl$:
                ◇ Write the system of equations of degree $q$ corresponding to the
                   keystream indexed from $x_j$
                ◇ Solve the system of equations by running MiniSAT on it.
                ◇ Run the SSG forward on the candidate(s) key(s).
                ◇ If the candidate key is the right one, output it and break the loop.
```

Now let us compute the amount of keystream necessary for this attack. We have computed the probability that all the l blocks have length at most $2k$, that is $(p_{q,k})^l$. Thus the keystream length N should satisfy$(N - kl) \cdot (p_{q,k})^l \geq 1$ if we want to find at least one match pair between the real internal sequence and our guess. Then we must have:

$$N \geq \frac{1}{(p_{q,k})^l}$$

At each step we try $(k - q + 1)^l$ possibilities for the length of the blocks. As the worst case for this attack is a number of steps N, the worst case complexity is:

$$\left(\frac{k - q + 1}{\sum_{j=q}^{k} \frac{\binom{j-1}{q-1}}{2^j}} \right)^{\frac{n}{q+h}}$$

where h stands for $h_{q,min}$.

This complexity is true if the information obtained is not redundant. We made simulations by choosing a number of blocks of exactly $\lceil \frac{1}{(p_{q,k})^l} \rceil$ and we always obtained the right key. If the key space given by the SAT solver is larger, we just perform an exhaustive search at small scale.

Now we give the results of our computations. The details of the computations are in appendix C. In this Section, instead of choosing random keys for our simulations, we chose keys such that the blocks in the initial state of the LFSR have length at most k. To achieve this, when generating randomly each block inside the initial state, we test (once it has reached length k) whether the number of "1"s among the even bits is at least q. If not, we start again from the beginning of the block. When the number of "1"s is as expected, we do it for the following block, until we find a compliant key.

We try many such compliant keys in order to limit also the length of the other blocks in the sequence but when k is very small ($k = q+1$ or $k = q+2$) we could not achieve the real conditions of the attack due to our limited computational power. Of course the running time would be shorter in the exact case described in the attack as, as we can see it in table 10 and 11 (appendix C), the shorter the blocks are, the faster MiniSAT is for these system of equations.

In table 5 and table 6, we give the total complexities of our attacks, for different block lengths. The Hamming weight of the feedback polynomials are 5 for both LFSR state length 256 and 512. The memory requirements during the MiniSAT computations are never more than 100Mb for this systems.

Finally Table 7 provides a performance comparison between Mihaljević attack, Hell-Johansson attack and our new method, for various sizes of n and of

Table 5. Total complexity and data complexity for $n = 256$

	$k = q+1$		$k = q+2$		$k = q+3$		$k = q+4$	
	time	data	time	data	time	data	time	data
$q = 2$	$2^{146.2}$	2^{64}	$2^{154.2}$	$2^{34.6}$	$2^{170.9}$	$2^{19.2}$	$2^{181.4}$	$2^{10.7}$
$q = 3$	$2^{151.4}$	$2^{79.3}$	$2^{147.2}$	$2^{47.3}$	2^{150}	$2^{28.7}$	$2^{157.2}$	$2^{17.5}$
$q = 4$	$2^{153.6}$	$2^{92.6}$	$2^{146.3}$	2^{59}	$2^{147.2}$	$2^{38.3}$	$2^{151.5}$	2^{25}

Table 6. Total time complexity and data complexity for $n = 512$

	$k = q+1$		$k = q+2$		$k = q+3$		$k = q+4$	
	time	data	time	data	time	data	time	data
$q = 2$	$2^{279.2}$	2^{128}	$2^{295.7}$	$2^{69.2}$	$2^{318.8}$	$2^{38.3}$	$2^{343.8}$	$2^{21.4}$
$q = 3$	$2^{277.4}$	$2^{158.7}$	$2^{269.6}$	$2^{94.6}$	$2^{279.3}$	$2^{57.5}$	$2^{293.5}$	2^{35}
$q = 4$	$2^{284.9}$	2^{185}	$2^{278.1}$	$2^{118.1}$	$2^{268.8}$	$2^{76.7}$	$> 2^{293}$	$2^{49.9}$

Table 7. Time complexity comparisons between Mihaljević, Hell *et al.* and our attack for the same data complexities

	$n = 256$				$n = 512$			
data	$2^{65.3}$	$2^{49.2}$	$2^{39.1}$	$2^{17.5}$	2^{128}	$2^{94.6}$	$2^{57.5}$	$2^{38.6}$
Mihaljević attack	2^{153}	2^{160}	$2^{165.5}$	2^{182}	2^{297}	2^{311}	2^{331}	2^{335}
Hell *et al* attack	$2^{160.2}$	$2^{164.8}$	$2^{167.8}$	$2^{176.4}$	2^{300}	$2^{308.3}$	2^{320}	2^{328}
Our attack	$2^{146.2}$	$2^{147.2}$	$2^{147.2}$	$2^{157.2}$	$2^{268.8}$	$2^{268.8}$	$2^{279.3}$	$2^{293.5}$

the amount of available keystream. For our attack, the results are bounded by our computational power and would have probably been better if we could have performed all the computations for $q = 4$ and $n = 512$. Anyway the obtained (heuristical) complexities show that for this feedback polynomial Hamming weight, our attack gives the best time/data tradeoff against the self-shrinking generator.

7 Conclusion

In [8] and [14], where the best known time/data tradeoffs are proposed on the self-shrinking generator, the authors show that their attack is independent from the value of the Hamming weight of the feedback polynomial defining the LFSR. However, the new algebraic guess-and-determine attack described here suggests that the security of SSG does depend on this Hamming weight. This new attack is very flexible concerning keystream requirement. As we use SAT solvers to solve our algebraic systems, it is not possible to compute a precise time complexity for our attack. However for small Hamming weight values (*i.e.* at most 5), this attack has a noticeably better complexity than the attacks of [8] and [14] and is even the best heuristical time/data tradeoff known so far on the self-shrinking generator.

Since Meier and Staffelbach original paper, avoiding low Hamming weight feedback polynomials has been a widely believed principle. However this rule did not materialize in previous recent attacks. With the new attacks described in this paper, we show explicitly that this principle remains true.

References

1. Bard, G.: Algorithms for Solving Linear and Polynomial Systems of Equations over Finite Fields, with Applications to Cryptanalysis. Ph.D. Dissertation, University of Maryland (2007)
2. Bard, G.V., Courtois, N.T., Jefferson, C.: Efficient Methods for Conversion and Solution of Sparse Systems of Low-Degree Multivariate Polynomials over GF(2) via SAT-Solvers (2007), `http://eprint.iacr.org/,/024`
3. Bard, G.V., Courtois, N.T.: Algebraic and Slide Attacks on KeeLoq. In: Preproceedings of FSE 2008, pp. 89-104 (2008)

4. Coppersmith, D., Krawczyk, H., Mansour, Y.: The Shrinking Generator. In: Stinson, D.R. (ed.) CRYPTO 1993. LNCS, vol. 773, pp. 22–39. Springer, Heidelberg (1994)
5. Courtois, N., Shamir, A., Patarin, J., Klimov, A.: Efficient Algorithms for solving Overdefined Systems of Multivariate Polynomial Equations. In: Preneel, B. (ed.) EUROCRYPT 2000. LNCS, vol. 1807, pp. 392–407. Springer, Heidelberg (2000)
6. Faugère, J.-C.: A new efficient algorithm for computing Gröbner bases (F_4). Journal of Pure and Applied Algebra 139, 61–88 (1999), www.elsevier.com/locate/jpaa
7. Faugère, J.-C.: A new efficient algorithm for computing Gröbner bases without reduction to zero (F5). In: Workshop on Applications of Commutative Algebra, Catania, Italy. ACM Press, New York (2002)
8. Hell, M., Johansson, T.: Two New Attacks on the Self-Shrinking Generator. IEEE Transactions on Information Theory 52(8), 3837–3843 (2006)
9. Krause, M.: BBD-based Cryptanalysis of Keystream Generators. In: Knudsen, L.R. (ed.) EUROCRYPT 2002. LNCS, vol. 2332, pp. 222–237. Springer, Heidelberg (2002)
10. Krawczyk, H.: Practical Aspects of the Shrinking Generator. In: Anderson, R. (ed.) FSE 1993. LNCS, vol. 809, pp. 45–46. Springer, Heidelberg (1994)
11. McDonald, C., Charnes, C., Pieprzyk, J.: Attacking Bivium with MiniS, AT (2007), http://eprint.iacr.org/2007/040
12. Meier, W., Staffelbach, O.: The Self-Shrinking Generator. In: De Santis, A. (ed.) EUROCRYPT 1994. LNCS, vol. 950, pp. 205–214. Springer, Heidelberg (1995)
13. Mihaljević, M.J.: A faster cryptanalysis of the self-shrinking generator. In: Pieprzyk, J.P., Seberry, J. (eds.) ACISP 1996. LNCS, vol. 1172, pp. 182–189. Springer, Heidelberg (1996)
14. Zhang, B., Feng, D.: New Guess-and-determine Attack on the Self-Shrinking Generator. In: Lai, X., Chen, K. (eds.) ASIACRYPT 2006. LNCS, vol. 4284, pp. 54–68. Springer, Heidelberg (2006)
15. Zenner, E., Krause, M., Lucks, S.: Improved Cryptanalysis of the Self-Shrinking Generator. In: Varadharajan, V., Mu, Y. (eds.) ACISP 2001. LNCS, vol. 2119, pp. 21–35. Springer, Heidelberg (2001)

A Computation of the Information Rate for the Self-Shrinking Generator

We have seen in Section 2.2 that the information rate that the keystream y reveals on the first m bits of internal sequence z is defined as

$$\alpha(m) = 1 - \frac{1}{m} H(Z^{(m)}|Y)$$

We have:

$$\begin{aligned} H(Z^{(m)}|Y) &= \sum_{y,z} \text{Proba}(Z^{(m)} = z, Y = y) \log(\text{Proba}(Z^{(m)} = z|Y = y)) \\ &= \sum_{y} \text{Proba}(Y = y) \sum_{z} \text{Proba}(Z^{(m)} = z|Y = y) \times \\ &\qquad\qquad \log(\text{Proba}(Z^{(m)} = z|Y = y)) \end{aligned}$$

The self-shrinking generator has the property that for $m \geq 1$ the probability that $C(z)$ is prefix for y for a randomly chosen and uniformly distributed $z \in \{0,1\}^m$ is the same for all keystream y. This implies that

$$\sum_z \text{Proba}(Z^{(m)} = z|Y = y) \log(\text{Proba}(Z^{(m)} = z|Y = y))$$

is the same for all y and

$$H(Z^{(m)}|Y) = \sum_z \text{Proba}(Z^{(m)} = z|Y = y) \log(\text{Proba}(Z^{(m)} = z|Y = y))$$

Let us call Z^0 the random variable of the first pair of bits of the internal sequence, Z^1 the second pair, etc. We have:

$$H(Z^0|Y) = \sum_{z_0} \text{Proba}(Z^0 = z_0|Y = y) \log(\text{Proba}(Z^0 = z_0|Y = y)) = \frac{3}{2}$$

Let us now show by recursion that $H(Z^{(2k)}|Y) = (\frac{3}{2})^k$.

$$\begin{aligned} H(Z^{(2k)}|Y) = \sum_{z_k,\cdots,z_0} & \text{Proba}(Z^k = z_k, \cdots, Z^0 = z_0|Y = y) \times \\ & \log(\text{Proba}(Z^k = z_k, \cdots, Z^0 = z_0|Y = y)) \end{aligned}$$

And as

$$\begin{aligned} &\text{Proba}(Z^k = z_k, \cdots, Z^0 = z_0|Y = y) = \\ &\text{Proba}(Z^k = z_k|Z^{k-1} = z_{k-1} \cdots, Z^0 = z_0, Y = y) \times \\ &\qquad \text{Proba}(Z^{k-1} = z_{k-1} \cdots, Z^0 = z_0|Y = y) \end{aligned}$$

we have:

$$\begin{aligned} H(Z^{(2k)}|Y) = & \sum_{z_{k-1},\cdots,z_0} \text{Proba}(Z^{k-1} = z_{k-1} \cdots, Z^0 = z_0|Y = y) \times \\ & \sum_{z_k} \text{Proba}(Z^k = z_k|Z^{k-1} = z_{k-1} \cdots, Z^0 = z_0, Y = y) \times \\ & \log(\text{Proba}(Z^k = z_k|Z^{k-1} = z_{k-1} \cdots, Z^0 = z_0, Y = y)) \\ + & \sum_{z_{k-1},\cdots,z_0} \text{Proba}(Z^{k-1} = z_{k-1}, \cdots, Z^0 = z_0|Y = y) \times \\ & \log(\text{Proba}(Z^{k-1} = z_{k-1}, \cdots, Z^0 = z_0|Y = y)) \times \\ & \sum_{z_k} \text{Proba}(Z^k = z_k|Z^{k-1} = z_{k-1} \cdots, Z^0 = z_0, Y = y). \end{aligned}$$

We know that

$$\sum_{z_k} \text{Proba}(Z^k = z_k|Z^{k-1} = z_{k-1} \cdots, Z^0 = z_0, Y = y) = 1$$

and by recursion

$$\sum_{z_{k-1},\cdots,z_0} \mathrm{Proba}(Z^{k-1} = z_{k-1}\cdots, Z^0 = z_0|Y = y) \times$$
$$\log(\mathrm{Proba}(Z^{k-1} = z_{k-1}\cdots, Z^0 = z_0|Y = y)) = \frac{3}{2}(k-1).$$

Once the first $k-1$ internal sequence pairs are fixed, let r be the number of 1s among the first bits of the $k-1$ pairs. Let us call y' the keystream sequence where the first r bits of y have been removed. Then the pair Z^k can be seen as the first pair of the internal sequence where $C(Z^k)$ is prefix for y'. Thus:

$$\mathrm{Proba}(Z^k = z_k|Z^{k-1} = z_{k-1}\cdots, Z^0 = z_0, Y = y) = \mathrm{Proba}(Z^k = z_k|Y = y')$$

and the first part of $H(Z^{(2k)}|Y)$ is

$$\sum_{z_k} \mathrm{Proba}(Z^k = z_k|Y = y') \log(\mathrm{Proba}(Z^k = z_k|Y = y')) = \frac{3}{2}.$$

We have obtained $H(Z^{(2k)}|Y) = \frac{3}{2}k$ and $\alpha(2k) = 1 - \frac{3}{2} \cdot \frac{1}{2k} \cdot k = \frac{1}{4}$.

B Equations for the General Case

This information can still be divided into three parts:

1. The first two bits of the block are known, this can be described by two linear equations.
2. The fact that the even bits of the subblock made of all the pairs of the block but the first one are all “0” but $q-1$ of them is described by :
   - $\binom{k-1}{q}$ degree q polynomials of the form $x_{2i_0}x_{2i_1}\cdots\ x_{2i_{q-1}} = 0$ where $2k$ is the length of the block and the x_{2i_j} are even bits of the subblock. This describes the fact that there is at most one “1” among the even bits of the subblock.
   - One equation of degree $q-1$: $\sum x_{i_0}x_{i_1}\cdots x_{i_{q-2}} = 1$, where the $x_{i_0}x_{i_1}\cdots\ x_{i_{q-2}}$ are all the monomials of degree $q-1$, describing the fact that there are at least $q-1$ “1”s among the even bits of the subblock.
3. The fact that the first keystream bit corresponding to this subblock follows the first even “1” of the subblock is described by $\binom{k-1}{q-1}$ degree q equations of the form $x_{2i_0}x_{2i_1}\cdots x_{2i_{q-2}}(x_{2i_0+1} + e_0) = 0$, the fact that the second keystream bit corresponding to this subblock follows the second even one of the subblock is described by $\binom{k-1}{q-1}$ degree q equations of the form $x_{2i_0}x_{2i_1}\cdots\ x_{2i_{q-2}}(x_{2i_1+1} + e_1) = 0$, etc, where the $e_0, e_1\cdots, e_{q-2}$ are the keystream bits corresponding to the subblock.

The known information on one block of length $2k$ is completely defined by the equations given above.

C Simulations Details

In tables 8 and 9, we give the complexity of the guess and the data complexity for our attack when the size of the LFSR is 256 or 512.

In tables 10 and 11, we give the time complexity of the MiniSAT solving part of the attack. We first give the running time in seconds, and then we give an estimation of the complexity of the form 2^a for each case to be able to compare our attack with the Hell and Johansson attack of [8]. This means that $2^a E$ is the running time of the solving, where En^3 would be the running time of the Gaussian elimination in the Hell and Johansson attack on the same machine. Concerning the Mihaljević attack, we just consider that testing the found key (by running the generator on it), is about n operations, where n is the size of the key.

We measured $E \approx 2^{-40}$ hours. With this convention, a running time of one hour corresponds to a complexity of 2^{40}.

Table 8. Complexity of the guess and data complexity for $n = 256$

	$k = q + 1$		$k = q + 2$		$k = q + 3$		$k = q + 4$	
	time	data	time	data	time	data	time	data
$q = 2$	2^{128}	2^{64}	2^{136}	$2^{34.6}$	$2^{147.2}$	$2^{19.2}$	$2^{159.3}$	$2^{10.7}$
$q = 3$	$2^{126.6}$	$2^{79.3}$	$2^{122.2}$	$2^{47.3}$	$2^{123.3}$	$2^{28.7}$	$2^{127.3}$	$2^{17.5}$
$q = 4$	2^{128}	$2^{92.6}$	$2^{119.8}$	2^{59}	2^{115}	$2^{38.3}$	2^{114}	2^{25}

Table 9. Complexity of the guess and data complexity for $n = 512$

	$k = q + 1$		$k = q + 2$		$k = q + 3$		$k = q + 4$	
	time	data	time	data	time	data	time	data
$q = 2$	2^{256}	2^{128}	2^{272}	$2^{69.2}$	$2^{294.3}$	$2^{38.3}$	$2^{318.6}$	$2^{21.4}$
$q = 3$	$2^{253.2}$	$2^{158.7}$	$2^{244.4}$	$2^{94.6}$	$2^{246.6}$	$2^{57.5}$	$2^{254.6}$	2^{35}
$q = 4$	2^{256}	2^{185}	$2^{239.6}$	$2^{118.1}$	2^{230}	$2^{76.7}$	2^{228}	$2^{49.9}$

Table 10. MiniSAT Computations for $n = 256$

	$k = q + 1$		$k = q + 2$		$k = q + 3$		$k = q + 4$	
$q = 2$	$< 0.001s$	$2^{18.2}$	$< 0.001s$	$2^{18.2}$	$0.046s$	$2^{23.7}$	$0.015s$	$2^{22.1}$
$q = 3$	$0.093s$	$2^{24.8}$	$0.109s$	2^{25}	$0.359s$	$2^{26.7}$	$3.39s$	$2^{29.9}$
$q = 4$	$0.171s$	$2^{25.6}$	0.311	$2^{26.5}$	15.6	$2^{32.2}$	$616s$	$2^{37.5}$

Table 11. MiniSAT Computations for $n = 512$

	$k = q + 1$		$k = q + 2$		$k = q + 3$		$k = q + 4$	
$q = 2$	$0.031s$	$2^{22.2}$	$0.046s$	$2^{23.7}$	$0.078s$	$2^{24.5}$	$0.125s$	$2^{25.2}$
$q = 3$	$0.06s$	$2^{29.2}$	$0.17s$	$2^{30.6}$	$22.3s$	$2^{37.7}$	$1641s$	$2^{38.9}$
$q = 4$	$1.171s$	$2^{28.9}$	$1308.5s$	$2^{38.5}$	1613	$2^{38.8}$	$> 24h$	$> 2^{45}$

New Form of Permutation Bias and Secret Key Leakage in Keystream Bytes of RC4

Subhamoy Maitra[1] and Goutam Paul[2]

[1] Applied Statistics Unit, Indian Statistical Institute,
Kolkata 700 108, India
subho@isical.ac.in

[2] Department of Computer Science and Engineering,
Jadavpur University, Kolkata 700 032, India
goutam_paul@cse.jdvu.ac.in

Abstract. Consider the permutation S in RC4. Roos pointed out in 1995 that after the Key Scheduling Algorithm (KSA) of RC4, each of the initial bytes of the permutation, i.e., $S[y]$ for small values of y, is biased towards some linear combination of the secret key bytes. In this paper, for the first time we show that the bias can be observed in $S[S[y]]$ too. Based on this new form of permutation bias after the KSA and other related results, a complete framework is presented to show that many keystream output bytes of RC4 are significantly biased towards several linear combinations of the secret key bytes. The results do not assume any condition on the secret key. We find new biases in the initial as well as in the 256-th and 257-th keystream output bytes. For the first time biases at such later stages are discovered without any knowledge of the secret key bytes. We also identify that these biases propagate further, once the information for the index j is revealed.

Keywords: Bias, Cryptanalysis, Keystream, Key Leakage, RC4, Stream Cipher.

1 Introduction

RC4 is one of the most well known stream ciphers. It has very simple implementation and is used in a number of commercial products till date. Being one of the popular stream ciphers, it has also been subjected to many cryptanalytic attempts for more than a decade. Though lots of weaknesses have already been explored in RC4 [1,2,3,4,5,6,7,8,10,11,12,13,15,16,17,19,20,21], it could not be thoroughly cracked yet and proper use of this stream cipher is still believed to be quite secure. This motivates the analysis of RC4.

The Key Scheduling Algorithm (KSA) and the Pseudo Random Generation Algorithm (PRGA) of RC4 are presented below. The data structure contains an array S of size N (typically, 256), which contains a permutation of the integers $\{0, \ldots, N-1\}$, two indices i, j and the secret key array K. Given a secret key k of l bytes (typically 5 to 16), the array K of size N is such that $K[y] = k[y \bmod l]$ for any y, $0 \leq y \leq N-1$.

K. Nyberg (Ed.): FSE 2008, LNCS 5086, pp. 253–269, 2008.

```
Algorithm KSA
Initialization:
    For $i = 0, \ldots, N-1$
        $S[i] = i$;
    $j = 0$;
Scrambling:
    For $i = 0, \ldots, N-1$
        $j = (j + S[i] + K[i])$;
        Swap($S[i], S[j]$);
```

```
Algorithm PRGA
Initialization:
    $i = j = 0$;
Output Keystream Generation Loop:
        $i = i + 1$;
        $j = j + S[i]$;
        Swap($S[i]$, $S[j]$);
        $t = S[i] + S[j]$;
        Output $z = S[t]$;
```

Apart from some minor details, the KSA and the PRGA are almost the same. In the KSA, the update of the index j depends on the secret key, whereas the key is not used in the PRGA. One may consider the PRGA as the KSA with all zero key. All additions in both the KSA and the PRGA are additions modulo N.

Initial empirical works based on the weaknesses of the RC4 KSA were explored in [17,21] and several classes of weak keys had been identified. In [17], experimental evidences of the bias of the initial permutation bytes after the KSA towards the secret key have been reported. It was also observed in [17] that the first keystream output byte of RC4 leaks information about the secret key when the first two secret key bytes add to 0 mod 256. A more general theoretical study has been performed in [11,12] which includes the observations of [17]. These biases do propagate to the keystream output bytes as observed in [5,11]. In [5], the Glimpse theorem [4] is used to show the propagation of biases in the initial keystream output bytes. On the other hand, a bias in the choice of index has been exploited in [11] to show a bias in the first keystream output byte.

More than a decade ago (1995), Roos [17] pointed out that the initial bytes $S[y]$ of the permutation after the KSA are biased towards some function f_y (see Section 1.1 for the definition of f_y) of the secret key. Since then several works [2,9,10,11,12,14] have considered biases of $S[y]$ either with functions of the secret key bytes or with absolute values and discussed applications of these biases. However, no research has so far been published to study how the bytes $S[S[y]]$ are related to the secret key for different values of y. Here we solve this problem, identifying substantial biases in this direction. It is interesting to note that as the KSA proceeds, the probabilities $P(S[y] = f_y)$ decrease monotonically, whereas the probabilities $P(S[S[y]] = f_y)$ first increases monotonically till the middle of the KSA and then decreases monotonically until the end of the KSA.

Using these results and other related techniques, we find new biases in the keystream output bytes towards the secret key. A complete framework is presented towards the leakage of information about the secret key in the keystream output bytes, that not only reveals new biases at a later stage (256, 257-th bytes), but also points out that the biases propagate further, once the information regarding j is known.

The works [2,7] also explain how secret key information is leaked in the keystream output bytes. In [2], it is considered that the first few bytes of the secret key is known and based on that the next byte of the secret key is predicted. The attack is based on how secret key information is leaked in the first keystream output byte of the PRGA. In [7], the same idea of [2] has been exploited with the Glimpse theorem [4] to find the information leakage about the secret key at the 257-th byte of the PRGA. Note that, our result is better than that of [7], as in [7] the bias is observed only when certain conditions on the secret key and IV hold. However, the biases we note at 256, 257-th bytes do not assume any such condition on the secret key.

1.1 Notations, Contributions and Outline

Let S_r be the permutation, i_r and j_r be the values of the indices i and j after r many rounds of the RC4 KSA, $1 \leq r \leq N$. Hence S_N is the permutation after the complete key scheduling. By S_0, we denote the initial identity permutation. During round r of the KSA, $i_r = r - 1$, $1 \leq r \leq N$, and hence the permutation S_r after round r can also be denoted by S_{i_r+1}.

Let S_r^G be the permutation, i_r^G and j_r^G be the values of the indices i and j, and z_r be the keystream output byte after r many rounds of the PRGA, $r \geq 1$. Clearly, $i_r^G = r \bmod N$. We also denote S_N by S_0^G as this is the permutation before the PRGA starts.

Further, let

$$f_y = \frac{y(y+1)}{2} + \sum_{x=0}^{y} K[x],$$

for $y \geq 0$. Note that all the additions and subtractions related to the key bytes, the permutation bytes and the indices are modulo N.

Our contribution can be summarized as follows.

- In Section 2, we present the results related to biased association of $S_N[S_N[y]]$ towards the linear combination f_y of the secret key bytes.
- In Section 3, we present a framework for identifying biases in RC4 keystream bytes towards several linear combinations of the secret key bytes.
  - In Section 3.1, we show that $P(z_N = N - f_0)$ is not a random association. This indicates bias at z_{256}.
  - In Section 3.2, we use the bias of $S_N[S_N[1]]$ (from Section 2) to prove that $P(z_{N+1} = N + 1 - f_1)$ is not a random association. This indicates bias at z_{257}.
  - In Section 3.3, we observe new biases in the initial keystream bytes apart from the known ones [5]. It is shown that for $3 \leq r \leq 32$, $P(z_r = f_{r-1})$ are not random associations.
  - These results are taken together in Section 3.4 to present cryptanalytic applications.
- In Section 4, considering that the values of index j are leaked at some points during the PRGA, we show that biases of the keystream output bytes towards the secret key are observed at a much later stage.

2 Bias of $S[S[y]]$ to Secret Key

We start this section discussing how $P(S_r[S_r[1]] = f_1)$ varies with round $r, 1 \leq r \leq N$, during the KSA of RC4. Once again, note that $f_1 = (K[0] + K[1] +$

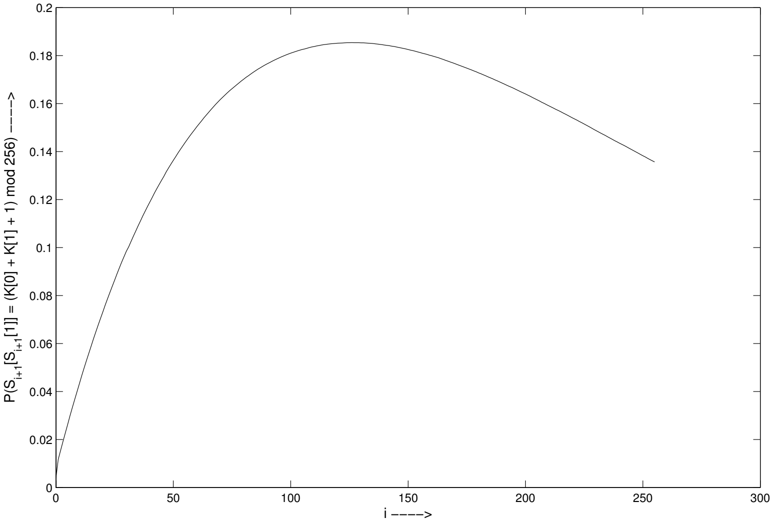


Fig. 1. $P(S_{i+1}[S_{i+1}[1]] = f_1)$ versus i $(r = i + 1)$ during RC4 KSA

1) mod N. To motivate, we like to refer to Figure 1 that demonstrates the nature of the curve with an experimentation using 10 million randomly chosen secret keys. The probability $P(S_r[S_r[1]] = f_1)$ increases till around $r = \frac{N}{2}$ where it gets the maximum value around 0.185 and then it decreases to 0.136 at $r = N$. Note that this nature is not similar to the nature of $P(S_r[1] = f_1)$ that decreases continuously as r increases during the KSA.

Towards the theoretical results, let us first present the base case for $r = 2$, i.e., after round 2 of the RC4 KSA.

Lemma 1. $P(S_2[S_2[1]] = K[0] + K[1] + 1) = \frac{3}{N} - \frac{4}{N^2} + \frac{2}{N^3}$.
Further, $P(S_2[S_2[1]] = K[0] + K[1] + 1 \wedge S_2[1] \leq 1) \approx \frac{2}{N}$.

Proof. The proof is based on three cases.

1. Let $K[0] \neq 0, K[1] = N - 1$. The probability of this event is $\frac{N-1}{N^2}$. Now $S_2[1] = S_1[K[0] + K[1] + 1] = S_1[K[0]] = S_0[0] = 0$. So, $S_2[S_2[1]] = S_2[0] = S_1[0] = K[0] = K[0]+K[1]+1$. Note that $S_2[0] = S_1[0]$, as $K[0]+K[1]+1 \neq 0$. Moreover, in this case, $S_2[1] \leq 1$.
2. Let $K[0] + K[1] = 0, K[0] \neq 1$, i.e., $K[1] \neq N - 1$. The probability of this event is $\frac{N-1}{N^2}$. Now $S_2[1] = S_1[K[0] + K[1] + 1] = S_1[1] = S_0[1] = 1$. Note that $S_1[1] = S_0[1]$, as $K[0] \neq 1$. So, $S_2[S_2[1]] = S_2[1] = 1 = K[0] + K[1] + 1$. Also, in this case, $S_2[1] \leq 1$.
3. $S_2[S_2[1]]$ could be $K[0] + K[1] + 1$ by random association except the two previous cases.

   Out of that, $S_2[1] \leq 1$ will happen in $\frac{2}{N}$ proportion of cases.

Thus $P(S_2[S_2[1]] = K[0] + K[1] + 1) = \frac{2(N-1)}{N^2} + (1 - \frac{2(N-1)}{N^2})\frac{1}{N} = \frac{3}{N} - \frac{4}{N^2} + \frac{2}{N^3}$. Further $P(S_2[S_2[1]] = K[0] + K[1] + 1 \wedge S_2[1] \leq 1) = \frac{2(N-1)}{N^2} + \frac{2}{N}(1 - \frac{2(N-1)}{N^2})\frac{1}{N} = \frac{2}{N} - \frac{4(N-1)}{N^4} \approx \frac{2}{N}$. □

Lemma 1 shows that after the second round ($i = 1, r = 2$), the event ($S_2[S_2[1]] = K[0] + K[1] + 1$) is not a random association.

Lemma 2. *Let* $p_r = P(S_r[S_r[1]] = K[0] + K[1] + 1 \wedge S_r[1] \leq r - 1)$ *for* $r \geq 2$. *Then for* $r \geq 3$, $p_r = (\frac{N-2}{N})p_{r-1} + \frac{1}{N} \cdot (\frac{N-2}{N}) \cdot (\frac{N-1}{N})^{2(r-2)}$.

Proof. After the $(r-1)$-th round is over, the permutation is S_{r-1}. In this case, $p_{r-1} = P(S_{r-1}[S_{r-1}[1]] = K[0] + K[1] + 1 \wedge S_{r-1}[1] \leq r - 2)$. The event $\big((S_r[S_r[1]] = K[0] + K[1] + 1) \wedge (S_r[1] \leq r - 1)\big)$ can occur in two mutually exclusive and exhaustive ways: $\big((S_r[S_r[1]] = K[0] + K[1] + 1) \wedge (S_r[1] \leq r - 2)\big)$ and $\big((S_r[S_r[1]] = K[0]+K[1]+1) \wedge (S_r[1] = r-1)\big)$. We compute the contribution of each separately.

In the r-th round, $i = r - 1$ and hence does not touch the indices $0, \ldots, r - 2$. Thus, the event $\big((S_r[S_r[1]] = K[0] + K[1] + 1) \wedge (S_r[1] \leq r - 2)\big)$ occurs if we already had $\big((S_{r-1}[S_{r-1}[1]] = K[0] + K[1] + 1) \wedge (S_{r-1}[1] \leq r - 2)\big)$ and $j_r \notin \{1, r - 1\}$. Thus, the contribution of this part is $p_{r-1}(\frac{N-2}{N})$.

The event $\big((S_r[S_r[1]] = K[0] + K[1] + 1) \wedge (S_r[1] = r - 1)\big)$ occurs if after the $(r-1)$-th round, $S_{r-1}[r-1] = r - 1$, $S_{r-1}[1] = K[0] + K[1] + 1$ and $j_r = 1$ causing a swap involving the indices 1 and $r - 1$.

1. We have $S_{r-1}[r-1] = r - 1$ if the location $r - 1$ is not touched during the rounds $i = 0, \ldots, r - 2$. This happens with a probability at least $(\frac{N-1}{N})^{r-1}$.
2. The event $S_{r-1}[1] = K[0]+K[1]+1$ may happen as follows. In the first round (when $i = 0$), $j_1 \notin \{1, K[0] + K[1] + 1\}$ so that $S_1[1] = 1$ and $S_1[K[0] + K[1]+1] = K[0]+K[1]+1$ with probability $(\frac{N-2}{N})$. After this, in the second round (when $i = 1$), we will have $j_2 = j_1 + S_1[1] + K[1] = K[0] + K[1] + 1$, and so after the swap, $S_2[1] = K[0]+K[1]+1$. Now, $K[0]+K[1]+1$ remains in location 1 from the end of round 2 till the end of round $(r-1)$ (when $i = r-2$) with probability $(\frac{N-1}{N})^{r-3}$. Thus, $P(S_{r-1}[1] = K[0] + K[1] + 1) = (\frac{N-2}{N}) \cdot (\frac{N-1}{N})^{r-3}$.
3. In the r-th round (when $i = r - 1$), j_r becomes 1 with probability $\frac{1}{N}$.

Thus, $P\big((S_r[S_r[1]] = K[0] + K[1] + 1) \wedge (S_r[1] = r - 1)\big) = (\frac{N-1}{N})^{r-1} \cdot (\frac{N-2}{N}) \cdot (\frac{N-1}{N})^{r-3} \cdot \frac{1}{N} = \frac{1}{N} \cdot (\frac{N-2}{N}) \cdot (\frac{N-1}{N})^{2(r-2)}$.

Adding the above two contributions, we get $p_r = (\frac{N-2}{N})p_{r-1} + \frac{1}{N} \cdot (\frac{N-2}{N}) \cdot (\frac{N-1}{N})^{2(r-2)}$. □

The recurrence in Lemma 2 along with the base case in Lemma 1 completely specify the probabilities p_r for all $r \in$[2,...,N].

Theorem 1. *After the complete KSA,*
$P(S_N[S_N[1]] = K[0] + K[1] + 1) \approx (\frac{N-1}{N})^{2(N-1)}$.

Proof. Using the approximation $\frac{N-2}{N} \approx (\frac{N-1}{N})^2$, the recurrence in Lemma 2 can be rewritten as $p_r = ap_{r-1} + a^{r-1}b$, where $a = (\frac{N-1}{N})^2$ and $b = \frac{1}{N}$. The solution of this recurrence is given by $p_r = a^{r-2}p_2 + (r-2)a^{r-1}b$, $r \geq 2$. Substituting the values of p_2 (from Lemma 1), a and b, we get $p_r = \frac{2}{N}(\frac{N-1}{N})^{2(r-2)} + (\frac{r-2}{N})(\frac{N-1}{N})^{2(r-1)}$. Substituting $r = N$ and noting the fact that $P\big((S_N[S_N[1]] = K[0] + K[1] + 1) \wedge (S_N[1] \leq N-1)\big) = P(S_N[S_N[1]] = K[0] + K[1] + 1)$, we get $P(S_N[S_N[1]] = K[0] + K[1] + 1) = \frac{2}{N}(\frac{N-1}{N})^{2(N-2)} + (\frac{N-2}{N})(\frac{N-1}{N})^{2(N-1)}$. Note that the second term (≈ 0.1348 for $N = 256$) dominates over the negligibly small first term (≈ 0.0011 for $N = 256$), and so $P(S_N[S_N[1]] = K[0] + K[1] + 1) \approx (\frac{N-1}{N})^{2(N-1)}$ (replacing $\frac{N-2}{N} = 1 - \frac{2}{N}$ by 1 in the second term). □

Now we like to present a more detailed observation. In [17,12], the association between $S_N[y]$ and f_y is shown. As we have observed the non-random association between $S_N[S_N[1]]$ and f_1, it is important to study what is the association between $S_N[S_N[y]]$ and f_y, and moving further, the association between $S_N[S_N[S_N[y]]]$ and f_y, for $0 \leq y \leq N-1$ and so on. Our experimental observations show that these associations are not random (i.e., much more than $\frac{1}{N}$) for initial values of y. The experimental observations (over 10 million runs of randomly chosen keys) are presented in Figure 2.

The theoretical analysis of the biases of $S_r[S_r[y]]$ towards f_y for small values of y is presented in Appendix A. The results involved in the process are tedious and we need to approximate certain quantities to get the following closed form formula.

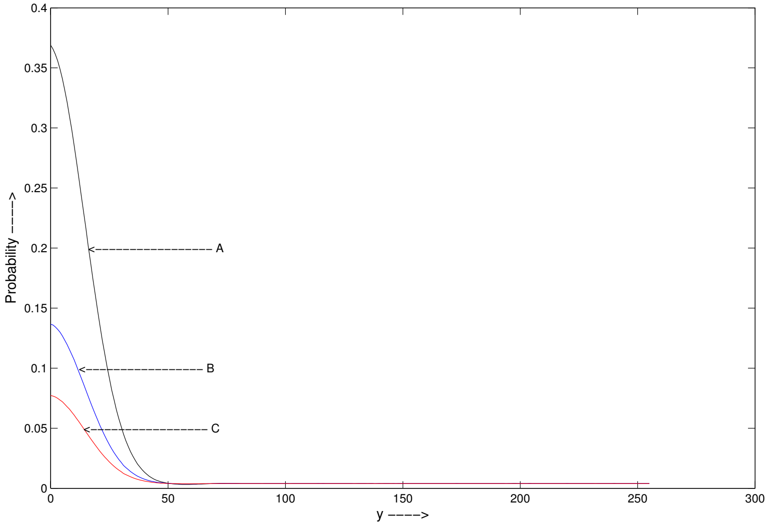


Fig. 2. A: $P(S_N[y] = f_y)$, B: $P(S_N[S_N[y]] = f_y)$, C: $P(S_N[S_N[S_N[y]]] = f_y)$ versus y $(0 \leq y \leq 255)$

Theorem 2. *After the complete KSA,* $P(S_N[S_N[y]] = f_y) \approx \frac{y}{N} \cdot (\frac{N-1}{N})^{\frac{y(y+1)}{2}+2(N-2)} + \frac{1}{N} \cdot (\frac{N-1}{N})^{\frac{y(y+1)}{2}-y+2(N-1)} + (\frac{N-y-1}{N}) \cdot (\frac{N-y}{N}) \cdot (\frac{N-1}{N})^{\frac{y(y+1)}{2}+2N-3}$, $0 \leq y \leq 31$.

Extending similar techniques, the association between $S_N[S_N \ldots [S_N[y]] \ldots]$ and f_y can be studied in general. Though the general results are combinatorially interesting, it is not immediate how they will be applicable to find further weaknesses in the RC4 PRGA. In terms of cryptanalytic point of view, we use the non-random association of $S_N[S_N[1]]$ relating f_1 (Theorem 1) to obtain the bias at the 257-th keystream output byte in Section 3.2.

3 New Biases in RC4 Keystream

We will first build the framework and then present new biases in Sections 3.1, 3.2 and 3.3, which were not known earlier.

Let us consider the existing result that relates each permutation byte after the KSA with certain linear combination of the secret key bytes.

Proposition 1. *[12, Theorem 1] Consider that the index j takes its values uniformly at random during the KSA rounds. Then,* $P(S_r[y] = f_y) \approx (\frac{N-y}{N}) \cdot (\frac{N-1}{N})^{[\frac{y(y+1)}{2}+r]} + \frac{1}{N}$, $0 \leq y \leq r-1$, $1 \leq r \leq N$.

Substituting $r = N$ in the statement of the above Proposition, we get the following.

Corollary 1. *The bias of the final permutation after the KSA towards the secret key is given by* $P(S_N[y] = f_y) = (\frac{N-y}{N}) \cdot (\frac{N-1}{N})^{[\frac{y(y+1)}{2}+N]} + \frac{1}{N}$, $0 \leq y \leq N-1$.

As explained in [12], the above result indicates significant biases for small values of y (more precisely, for $0 \leq y \leq 47$), that is supported by the experimental result presented in [17].

The Glimpse Main Theorem [4,7] states that after the r-th round of the PRGA, $r \geq 1$, $P(S_r^G[j_r^G] = r - z_r) = P(S_r^G[i_r^G] = j_r^G - z_r) = \frac{2}{N}$. We rewrite the first relation between $S_r^G[j_r^G]$ and $r - z_r$ as the following proposition.

Proposition 2. $P(z_r = r - S_{r-1}^G[i_r^G]) = \frac{2}{N}$, $r \geq 1$.

Proof. $S_r^G[j_r^G]$ is assigned the value of $S_{r-1}^G[i_r^G]$. As the Glimpse Main Theorem gives $P(z_r = r - S_r^G[j_r^G]) = \frac{2}{N}$ for $r \geq 1$, we get $P(z_r = r - S_{r-1}^G[i_r^G]) = \frac{2}{N}$ for $r \geq 1$. □

The idea of writing the Glimpse Main Theorem in the form of Proposition 2 is due to the fact that relating "z_r to $S_{r-1}^G[i_r^G]$" will ultimately relate "z_r to the secret key bytes", as the permutations in the initial rounds of the PRGA are related to the secret key.

Now we start with our results. The following lemma shows how the permutation bytes at rounds t and $r-1$ of the PRGA, for $t + 2 \leq r$, are related.

Lemma 3. *Let $P(S_t^G[i_r^G] = X) = q_{t,r}$, for some X. Then, for $t+2 \le r \le t+N$, $P(S_{r-1}^G[i_r^G] = X) = q_{t,r} \cdot \left[(\frac{N-1}{N})^{r-t-1} - \frac{1}{N}\right] + \frac{1}{N}$.*

Proof. We consider two separate cases.

1. $S_t^G[i_r^G] = X$ and during the next $(r-t-1)$ rounds of the PRGA, the index i_r^G is not touched by any of the $r-t-1$ many j values (since $t+2 \le r \le t+N$, the index i_r^G is not touched by any of the $r-t-1$ many i values anyway). The first event occurs with probability $q_{t,r}$ and the second event occurs with probability $(\frac{N-1}{N})^{r-t-1}$. Thus the contribution of this case is $q_{t,r} \cdot (\frac{N-1}{N})^{r-t-1}$.
2. $S_t^G[i_r^G] \neq X$ and still $S_{r-1}^G[i_r^G]$ equals X by random association. The contribution of this case is $(1 - q_{t,r}) \cdot \frac{1}{N}$.

Thus, adding the above two contributions, we get $P(S_{r-1}^G[i_r^G] = X) = q_{t,r} \cdot (\frac{N-1}{N})^{r-t-1} + (1 - q_{t,r}) \cdot \frac{1}{N} = q_{t,r} \cdot \left[(\frac{N-1}{N})^{r-t-1} - \frac{1}{N}\right] + \frac{1}{N}$. □

Remark 1. The above result holds for $t+2 \le r \le t+N$, and not for $r = t+1$. If we take $r = t+1$, then $S_{r-1}^G = S_t^G$, which is our starting point, i.e., $P(S_{r-1}^G[i_r^G] = X) = P(S_t^G[i_r^G] = X) = q_{t,r}$.

The following is an immediate consequence of Lemma 3.

Corollary 2. *For $2 \le r \le N-1$, $P(S_{r-1}^G[r] = f_r) = \left[(\frac{N-r}{N}) \cdot (\frac{N-1}{N})^{[\frac{r(r+1)}{2}+N]} + \frac{1}{N}\right] \cdot \left[(\frac{N-1}{N})^{r-1} - \frac{1}{N}\right] + \frac{1}{N}$.*

Proof. For $2 \le r \le N-1$, we have $i_r^G = r$. Taking $X = f_r$ and $t = 0$ in Lemma 3, we have $q_{0,r} = P(S_0^G[r] = f_r) = P(S_N[r] = f_r) = (\frac{N-r}{N}) \cdot (\frac{N-1}{N})^{[\frac{r(r+1)}{2}+N]} + \frac{1}{N}$ (by Corollary 1), and hence $P(S_{r-1}^G[r] = f_r) = \left[(\frac{N-r}{N}) \cdot (\frac{N-1}{N})^{[\frac{r(r+1)}{2}+N]} + \frac{1}{N}\right] \cdot \left[(\frac{N-1}{N})^{r-1} - \frac{1}{N}\right] + \frac{1}{N}$. □

Next, we present the bias of each keystream output byte to a combination of the secret key bytes in the following lemma.

Lemma 4. *Let $P(S_{r-1}^G[i_r^G] = f_{i_r^G}) = w_r$, for $r \ge 1$. Then $P(z_r = r - f_{i_r^G}) = \frac{1}{N} \cdot (1 + w_r)$, $r \ge 1$.*

Proof. We consider two separate cases in which the event $(z_r = r - f_{i_r^G})$ can occur.

1. $S_{r-1}^G[i_r^G] = f_{i_r^G}$ and $z_r = r - S_{r-1}^G[i_r^G]$. The contribution of this case is $P(S_{r-1}^G[i_r^G] = f_{i_r^G}) \cdot P(z_r = r - S_{r-1}^G[i_r^G]) = w_r \cdot \frac{2}{N}$ (by Proposition 2).
2. $S_{r-1}^G[i_r^G] \neq f_{i_r^G}$, and still $z_r = r - f_{i_r^G}$ due to random association. So the contribution of this case is $P(S_{r-1}^G[i_r^G] \neq f_{i_r^G}) \cdot \frac{1}{N} = (1 - w_r) \cdot \frac{1}{N}$.

Adding the above two contributions, we get the total probability as $w_r \cdot \frac{2}{N} + (1 - w_r) \cdot \frac{1}{N} = \frac{1}{N} \cdot (1 + w_r)$. □

Some results for biases in initial keystream bytes has earlier been pointed out in [5] that has later been discussed in [19] too. We detail these biases giving explicit formula under our theoretical framework.

Theorem 3.
(1) $P(z_1 = 1 - f_1) = \frac{1}{N} \cdot \Big(1 + (\frac{N-1}{N})^{N+2} + \frac{1}{N}\Big)$.
(2) *For* $2 \le r \le N-1$,
$P(z_r = r - f_r) = \frac{1}{N} \cdot \Big(1 + \big[(\frac{N-r}{N}) \cdot (\frac{N-1}{N})^{[\frac{r(r+1)}{2}+N]} + \frac{1}{N}\big] \cdot \big[(\frac{N-1}{N})^{r-1} - \frac{1}{N}\big] + \frac{1}{N}\Big)$.

Proof. First, we prove part (1). In the first round, i.e., when $r = 1$, we have $i_r^G = 1$ and $f_{i_r^G} = f_1$, and so $w_1 = P(S_0^G[1] = f_1) = P(S_N[1] = f_1) = (\frac{N-1}{N}) \cdot (\frac{N-1}{N})^{[\frac{1(1+1)}{2}+N]} + \frac{1}{N} = (\frac{N-1}{N})^{N+2} + \frac{1}{N}$ (by Corollary 1). Now, using Lemma 4, we get $P(z_1 = 1 - f_1) = \frac{1}{N} \cdot (1 + w_1) = \frac{1}{N} \cdot \Big(1 + (\frac{N-1}{N})^{N+2} + \frac{1}{N}\Big)$.

Next, we prove part (2). From Corollary 2, $w_r = P(S_{r-1}^G[r] = f_r) = \big[(\frac{N-r}{N}) \cdot (\frac{N-1}{N})^{[\frac{r(r+1)}{2}+N]} + \frac{1}{N}\big] \cdot \big[(\frac{N-1}{N})^{r-1} - \frac{1}{N}\big] + \frac{1}{N}$, $2 \le r \le N-1$. Now, using Lemma 4, we get $P(z_r = r - f_r) = \frac{1}{N} \cdot (1 + w_r) = \frac{1}{N} \cdot \Big(1 + \big[(\frac{N-r}{N}) \cdot (\frac{N-1}{N})^{[\frac{r(r+1)}{2}+N]} + \frac{1}{N}\big] \cdot \big[(\frac{N-1}{N})^{r-1} - \frac{1}{N}\big] + \frac{1}{N}\Big)$. □

Note that Lemma 3 or Corollary 2 is not used in proving part (1) of the above theorem. It is proved directly from Corollary 1. In fact, Lemma 3 can not be used in part (1), as here we have $r = t + 1$ with $t = 0$ (see Remark 1).

To have a clear understanding of the quantity of the biases, Table 1 lists the numerical values of the probabilities according to the formula given in Theorem 3. Note that the random association is $\frac{1}{N}$, which is 0.0039 for $N = 256$.

Close to the round 48, the biases tend to disappear. This is indicated by the convergence of the values to the probability $\frac{1}{256} = 0.0039$.

Table 1. The probabilities computed following Theorem 3

r	$P(z_r = r - f_r)$							
1-8	0.0053	0.0053	0.0053	0.0053	0.0052	0.0052	0.0052	0.0051
9-16	0.0051	0.0050	0.0050	0.0049	0.0048	0.0048	0.0047	0.0047
17-24	0.0046	0.0046	0.0045	0.0045	0.0044	0.0044	0.0043	0.0043
25-32	0.0043	0.0042	0.0042	0.0042	0.0041	0.0041	0.0041	0.0041
33-40	0.0041	0.0040	0.0040	0.0040	0.0040	0.0040	0.0040	0.0040
41-48	0.0040	0.0040	0.0040	0.0040	0.0040	0.0039	0.0039	0.0039

One may check that $P(z_1 = 1 - f_1) = \frac{1}{N}(1 + 0.36)$ and that decreases to $P(z_{32} = 32 - f_{32}) = \frac{1}{N}(1 + 0.05)$, but still then it is 5% more than the random association.

3.1 Bias in the 256-th Keystream Output Byte

Interestingly, the biases again reappear after rounds 256 and 257. First we present the bias for the 256-th keystream byte.

Theorem 4. $P(z_N = N - f_0) = \frac{1}{N} \cdot \Big(1 + (\frac{N-1}{N})^{2N-1} + \frac{1}{N^2} \cdot (\frac{N-1}{N})^{N-1} - \frac{1}{N^2} + \frac{1}{N}\Big)$.

Proof. During the N-th round of the PRGA, $i_N^G = N \bmod N = 0$. Taking $X = f_0$, $t = 0$ and $r = N$ in Lemma 3, we have $q_{0,N} = P(S_0^G[0] = f_0) = P(S_N[0] = f_0) = (\frac{N-1}{N})^N + \frac{1}{N}$ (by Corollary 1), and hence $w_N = P(S_{N-1}^G[0] = f_0) = \big[(\frac{N-1}{N})^N + \frac{1}{N}\big] \cdot \big[(\frac{N-1}{N})^{N-1} - \frac{1}{N}\big] + \frac{1}{N} = (\frac{N-1}{N})^{2N-1} + \frac{1}{N^2} \cdot (\frac{N-1}{N})^{N-1} - \frac{1}{N^2} + \frac{1}{N}$. Thus, by Lemma 4, the bias is given by $P(z_N = N - f_0) = \frac{1}{N} \cdot (1 + w_N) = \frac{1}{N} \cdot \Big(1 + (\frac{N-1}{N})^{2N-1} + \frac{1}{N^2} \cdot (\frac{N-1}{N})^{N-1} - \frac{1}{N^2} + \frac{1}{N}\Big)$. □

For $N = 256$, $w_N = w_{256} = 0.1392$ and the bias turns out to be 0.0045 (i.e., $\frac{1}{256}(1 + 0.1392)$). Experimental results confirm this bias.

3.2 Bias in the 257-th Keystream Output Byte

We will now show that the bias in the 257-th keystream output byte follows from Theorem 1, i.e., $P(S_N[S_N[1]] = K[0] + K[1] + 1) \approx (\frac{N-1}{N})^{2(N-1)}$.

Theorem 5. $P(z_{N+1} = N + 1 - f_1)$
$= \frac{1}{N} \cdot \Big(1 + (\frac{N-1}{N})^{3(N-1)} - \frac{1}{N} \cdot (\frac{N-1}{N})^{2(N-1)} + \frac{1}{N}\Big)$.

Proof. During the $(N+1)$-th round, we have, $i_{N+1}^G = (N+1) \bmod N = 1$. Taking $X = f_1$, $t = 1$ and $r = N + 1$ in Lemma 3, we have $q_{1,N+1} = P(S_1^G[1] = f_1) = P(S_N[S_N[1]] = f_1) = (\frac{N-1}{N})^{2(N-1)}$, and hence $w_{N+1} = P(S_N^G[1] = f_1) = (\frac{N-1}{N})^{2(N-1)} \cdot \big[(\frac{N-1}{N})^{N-1} - \frac{1}{N}\big] + \frac{1}{N} = (\frac{N-1}{N})^{3(N-1)} - \frac{1}{N} \cdot (\frac{N-1}{N})^{2(N-1)} + \frac{1}{N}$. Now, using Lemma 4, we get $P(z_{N+1} = N + 1 - f_1) = \frac{1}{N} \cdot (1 + w_{N+1}) = \frac{1}{N} \cdot \Big(1 + (\frac{N-1}{N})^{3(N-1)} - \frac{1}{N} \cdot (\frac{N-1}{N})^{2(N-1)} + \frac{1}{N}\Big)$. □

For $N = 256$, $w_{N+1} = w_{257} = 0.0535$ and $P(z_{257} = 257 - f_1) = \frac{1}{N} \cdot (1 + 0.0535) = 0.0041$ which also conforms to experimental observation.

3.3 More Biases in Initial Bytes of RC4 Keystream

The biases of z_r with $r - f_r$ for the initial keystream output bytes have been pointed out in Theorem 3. Interestingly, experimental observation reveals bias of z_r with f_{r-1} too. The results are presented in Table 2 which is experimented over hundred million (10^8) randomly chosen keys of 16 bytes. For proper random association, $P(z_r = f_{r-1})$ should have been $\frac{1}{256}$, i.e., 0.0039.

Following our experimental observation, the explanation of the fact $P(z_3 = f_2) > \frac{1}{256}$ was pointed out in [18]. We present the idea of [18] in this paragraph. Assume that after the third round of the KSA, $S_3[2]$ takes the value f_2, and is hit by j later in the KSA. Then f_2 is swapped with $S_k[k]$ and consider that $S_k[k]$ has remained k so far. Further, suppose that $S_N[3] = 0$ holds. Thus, $S_N[2] = k$, $S_N[k] = f_2$ and $S_N[3] = 0$ at the end of the KSA. In the second round of the PRGA, $S_1^G[2] = k$ is swapped with a more or less random location $S_1^G[l]$. Therefore, $S_2^G[l] = k$ and $j_2^G = l$. In the next round, $i = 3$ and points to

Table 2. Additional bias of the keystream bytes towards the secret key

r	$P(z_r = f_{r-1})$							
1-8	0.0043	0.0039	0.0044	0.0044	0.0044	0.0044	0.0043	0.0043
9-16	0.0043	0.0043	0.0043	0.0042	0.0042	0.0042	0.0042	0.0042
17-24	0.0041	0.0041	0.0041	0.0041	0.0041	0.0040	0.0040	0.0040
25-32	0.0040	0.0040	0.0040	0.0040	0.0040	0.0040	0.0040	0.0040
33-40	0.0039	0.0039	0.0039	0.0039	0.0039	0.0039	0.0039	0.0039
41-48	0.0039	0.0039	0.0039	0.0039	0.0039	0.0039	0.0039	0.0039

$S_2^G[3] = 0$. So j does not change and hence $j_3^G = l = j_2^G$. Thus, $S_2^G[l] = k$ is swapped with $S_2^G[3] = 0$, and one gets $S_3^G[l] = 0$ and $S_3^G[3] = k$. The output z_3 is now $S_3^G[S_3^G[i] + S_3^G[j_3^G]] = S_3^G[k + 0] = S_3^G[k] = f_2$.

Along the same line of arguments given in [18], we here provide a detailed proof considering the event $z_r = f_{r-1}$ for $r > 2$ in general and explicitly derive a formula for $P(z_r = f_{r-1})$. The proof depends on $P(S_N[r] = 0)$ for different r values. The explicit formula for the probabilities $P(S_N[u] = v)$ was derived for the first time in [9] and the problem was addressed again in [10,14].

Proposition 3. *[14, Theorem 1, Item 2] For* $0 \le v \le N-1$, $v \le u \le N-1$, $P(S_N[u] = v) = \frac{1}{N} \cdot (\frac{N-1}{N})^{N-1-u} + \frac{1}{N} \cdot (\frac{N-1}{N})^{v+1} - \frac{1}{N} \cdot (\frac{N-1}{N})^{N+v-u}$.

Theorem 6. *For* $3 \le r \le N$, $P(z_r = f_{r-1}) = (\frac{N-1}{N}) \cdot (\frac{N-r}{N}) \cdot \Big((\frac{N-r+1}{N}) \cdot (\frac{N-1}{N})^{[\frac{r(r-1)}{2}+r]} + \frac{1}{N}\Big) \cdot (\frac{N-2}{N})^{N-r+1} \cdot (\frac{N-3}{N})^{r-2} \cdot \gamma_r + \frac{1}{N}$,
where $\gamma_r = \frac{1}{N} \cdot (\frac{N-1}{N})^{N-1-r} + \frac{1}{N} \cdot (\frac{N-1}{N}) - \frac{1}{N} \cdot (\frac{N-1}{N})^{N-r}$.

Proof. Substituting $y = r-1$ in Proposition 1, we have $P(S_r[r-1] = f_{r-1}) = \alpha_r$, where $\alpha_r \approx (\frac{N-r+1}{N}) \cdot (\frac{N-1}{N})^{[\frac{r(r-1)}{2}+r]} + \frac{1}{N}$, $1 \le r \le N$. After round r, suppose that the index $r-1$ is touched for the first time by j_{t+1} in round $t+1$ of the KSA and due to the swap the value f_{r-1} is moved to the index t, $r \le t \le N-1$ and also prior to this swap the value at the index t was t itself, which now comes to the index $r-1$. This means that from round $r+1$ to round t (both inclusive), the pseudo-random index j has not taken the values $r-1$ and t. So, after round $t+1$, $P\big((S_{t+1}[r-1] = t) \wedge (S_{t+1}[t] = f_{r-1})\big)$
$= P\big((S_t[r-1] = f_{r-1}) \wedge (S_t[t] = t) \wedge (j_{t+1} = r-1)\big)$
$= \alpha_r \cdot (\frac{N-2}{N})^{t-r} \cdot \frac{1}{N}$.
From round $t+1$ until the end of the KSA, f_{r-1} remains in index t and t remains in index $r-1$ with probability $(\frac{N-2}{N})^{N-t}$. Thus,
$P\big((S_N[r-1] = t) \wedge (S_N[t] = f_{r-1})\big)$
$= \alpha_r \cdot (\frac{N-2}{N})^{t-r} \cdot \frac{1}{N} \cdot (\frac{N-2}{N})^{N-t}$
$= \alpha_r \cdot (\frac{N-2}{N})^{N-r} \cdot \frac{1}{N} = \beta_r$ (say). Also, from Proposition 3, we have $P(S_N[r] = 0) = \gamma_r$, where $\gamma_r = \frac{1}{N} \cdot (\frac{N-1}{N})^{N-1-r} + \frac{1}{N} \cdot (\frac{N-1}{N}) - \frac{1}{N} \cdot (\frac{N-1}{N})^{N-r}$.

Suppose the indices $r-1$, t and r are not touched by the pseudo-random index j in the first $r-2$ rounds of the PRGA. This happens with probability $(\frac{N-3}{N})^{r-2}$. In round $r-1$ of the PRGA, due to the swap, the value t at index $r-1$

moves to the index j^G_{r-1} with probability 1, and $j^G_{r-1} \notin \{t, r\}$ with probability $(\frac{N-2}{N})$. Further, if $S^G_{r-1}[r]$ remains 0, then in round r of the PRGA, $j^G_r = j^G_{r-1}$ and $z_r = S^G_r\left[S^G_r[r] + S^G_r[j^G_r]\right] = S^G_r\left[S^G_{r-1}[r] + S^G_{r-1}[j^G_{r-1}]\right] = S^G_r[0+t] = S^G_r[t] = f_{r-1}$ with probability $\beta_r \cdot \gamma_r \cdot (\frac{N-3}{N})^{r-2} \cdot (\frac{N-2}{N}) = \delta_r$ (say). Since, t can values $r, r+1, r+2, \ldots, N-1$, the total probability is $\delta_r \cdot (N-r)$. Substituting the values of $\alpha_r, \beta_r, \gamma_r, \delta_r$, we get the probability that the event $(z_r = f_{r-1})$ occurs in the above path is $p = (\frac{N-r}{N}) \cdot \left((\frac{N-r+1}{N}) \cdot (\frac{N-1}{N})^{[\frac{r(r-1)}{2}+r]} + \frac{1}{N}\right) \cdot (\frac{N-2}{N})^{N-r+1} \cdot (\frac{N-3}{N})^{r-2} \cdot \gamma_r$.

If the above path is not followed, still there is $(1-p) \cdot \frac{1}{N}$ probability of occurrence of the event due to random association. Adding these two probabilities, we get the result. □

The theoretically computed values of the probabilities according to the above theorem match with the estimated values provided in Table 2. It will be interesting to justify the bias at $r = 1$ and the absence of the bias at $r = 2$ as observed experimentally in Table 2. These two cases are not covered in Theorem 6.

3.4 Cryptanalytic Applications

Here we accumulate the results explained above. Consider the first keystream output byte z_1 of the PRGA. We find the theoretical result that $P(z_1 = 1 - f_1) = 0.0053$ (see Theorem 3 and Table 1) and the experimental observation that $P(z_1 = f_0) = 0.0043$ (see Table 2). Further, from [11], we have the result that $P(z_1 = f_2) = 0.0053$. Taking them together, one can check that the $P(z_1 = f_0 \vee z_1 = 1 - f_1 \vee z_1 = f_2) = 1 - (1 - 0.0043) \cdot (1 - 0.0053) \cdot (1 - 0.0053) = 0.0148$. (The independence assumption in calculating the probability is supported by detailed experimentation with 100 different runs, each run presenting the average probability considering 10 million randomly chosen secret keys of 16 bytes.) Our result indicates that out of randomly chosen 10000 secret keys, in 148 cases on an average, z_1 reveals f_0 or $1 - f_1$ or f_2, i.e., $K[0]$ or $1 - (K[0] + K[1] + 1)$ or $(K[0] + K[1] + K[2] + 3)$. If, however, one tries a random association, considering that z_1 will be among three randomly chosen values v_1, v_2, v_3 from $[0, \ldots, 255]$, then $P(z_1 = v_1 \vee z_1 = v_2 \vee z_1 = v_3) = 1 - (1 - \frac{1}{256})^3 = 0.0117$. Thus one can guess z_1 with an additional advantage of $\frac{0.0148-0.0117}{0.0117} \cdot 100\% = 27\%$ over the random guess.

Looking at z_2, we have $P(z_2 = 2 - f_2) = 0.0053$ (see Theorem 3 and Table 1), which provides an advantage of $\frac{0.0053-0.0039}{0.0039} \cdot 100\% = 36\%$.

Similarly, referring to Theorem 3 and Theorem 6 (and also Table 1 and Table 2), significant biases can be observed in $P(z_r = f_{r-1} \vee z_r = r - f_r)$ for $r = 3$ to 32 over random association.

Now consider the following scenario with the events $E_1, \ldots, E_{32}$, where $E_1 : (z_1 = f_0 \vee z_1 = 1 - f_1 \vee z_1 = f_2)$, $E_2 : (z_2 = 2 - f_2)$, and $E_r : (z_r = f_{r-1} \vee z_r = r - f_r)$ for $3 \leq r \leq 32$. Observing the first 32 keystream output bytes $z_1, \ldots, z_{32}$, one may try to guess the secret key assuming that 3 or more of the events $E_1, \ldots, E_{32}$ occur. We experimented with 10 million randomly chosen secret keys of length 16 bytes. We found that 3 or more of the events occur in 0.0028 proportion of cases, which is true for 0.0020 proportion of cases for random association. This demonstrates a substantial advantage (40%) over random guess.

4 Further Biases When j Is Known During PRGA

In all the currently known biases as well as in all the new biases discussed in this paper so far, it is assumed that the value of the pseudo-random index j is unknown. In this section, we are going to show that the biases in the permutation at some stage of the PRGA propagates to the keystream output bytes at a later stage, if the value of the index j at the earlier stage is known.

Suppose that we know the value j_t^G of j after the round t in the PRGA. With high probability, the value V at the index j_t^G will remain there until j_t^G is touched by the deterministic index i for the first time after a few more rounds depending on what was the position of i at the t-th stage. This immediately leaks V in keystream output byte. More importantly, if the value V is biased to the secret key bytes, then that information will be leaked too.

Formally, let $P(S_t^G[j_t^G] = V) = \eta_t$ for some V. j_t^G will be touched by i in round r, where $r = j_t^G$ or $N + j_t^G$ depending on whether $j_t^G > t$ or $j_t^G \leq t$ respectively. By Lemma 3, we would have $P(S_{r-1}^G[j_t^G] = V) = \eta_t \cdot \left[(\frac{N-1}{N})^{r-t-1} - \frac{1}{N}\right] + \frac{1}{N}$. Now, Lemma 4 immediately gives $P(z_r = r - V) = \frac{1}{N} \cdot \left(1 + \eta_t \cdot \left[(\frac{N-1}{N})^{r-t-1} - \frac{1}{N}\right] + \frac{1}{N}\right)$.

For some special V's, the form of η_t may be known. In that case, it will be advantageous to probe the values of j at particular rounds. For example, according to Corollary 2, after the $(t-1)$-th round of the PRGA, $S_{t-1}^G[t]$ is biased to the linear combination f_t of the secret key bytes with probability $\eta_t = \left[(\frac{N-t}{N}) \cdot (\frac{N-1}{N})^{[\frac{t(t+1)}{2}+N]} + \frac{1}{N}\right] \cdot \left[(\frac{N-1}{N})^{t-1} - \frac{1}{N}\right] + \frac{1}{N}$. Now, at round t, f_t would move to the index j_t due to the swap, and hence $S_t^G[j_t]$ will be biased to f_t with the same probability. So, the knowledge of j_t will leak information about f_t in round j_t^G or $N + j_t^G$ according as $j_t^G > t$ or $j_t^G \leq t$ respectively.

If we know the values of j at multiple stages of the PRGA (it may be possible to read some values of j through side-channel attacks), then the biases propagate further down the keystream output bytes. The following example illustrates how the biases propagate down the keystream output bytes when single as well as multiple j^G values are known.

Example 1. Suppose we know the value of j_5^G as 18. With probability η_5, $S_4^G[5]$ would have remained f_5 which would move to index 18 due to the swap in round 5, i.e., $S_5^G[18] = f_5$. With approximately $\eta_5 \cdot \left[(\frac{N-1}{N})^{18-5-1} - \frac{1}{N}\right] + \frac{1}{N}$ probability, f_5 would remain in index 18 till the end of the round 18-1 = 17. So, we immediately get a bias of z_{18} with $18 - f_5$.

Moreover, in round 18, f_5 would move from index 18 to j_{18}^G. So, if the value of j_{18}^G is also known, say $j_{18}^G = 3$, then we have $S_{18}^G[3] = f_5$. We can apply the same line of arguments for round $256 + 3 = 259$ to get a bias of z_{259} with $259 - f_5$.

Experiments with 1 billion random keys demonstrate that in this case the bias of z_{18} towards $18 - f_5$ is 0.0052 and the bias of z_{259} towards $259 - f_5$ is 0.0044. These conform to the theoretical values and show that the knowledge of j during the PRGA helps in finding non-random association (away from $\frac{1}{256} = 0.0039$) between the keystream output bytes and the secret key.

5 Conclusion

In this paper, we present several new observations on weaknesses of RC4. It is shown that biases towards the secret key exists at the permutation bytes $S[S[y]]$ for different y values. To our knowledge, this is the first attempt to formally analyze the biases of $S[S[y]]$ and its implications towards the security of RC4. Moreover, a framework is built to analyze biases of the keystream output bytes towards different linear combinations of the secret key bytes. Subsequently, theoretical results are proved to show that RC4 keystream output bytes at the indices 1 to 32 leak significant information about the secret key bytes. We also discovered and proved new biases towards the secret key at the 256-th and the 257-th keystream output bytes. Moreover, we show that if one knows the value of j during some rounds of the PRGA, then the biases propagate further down the keystream.

Acknowledgment. The authors like to thank Mr. Snehasis Mukherjee, Indian Statistical Institute, Kolkata for his support in the preparation of the graphs.

References

1. Fluhrer, S.R., McGrew, D.A.: Statistical Analysis of the Alleged RC4 Keystream Generator. In: Schneier, B. (ed.) FSE 2000. LNCS, vol. 1978, pp. 19–30. Springer, Heidelberg (2001)
2. Fluhrer, S.R., Mantin, I., Shamir, A.: Weaknesses in the Key Scheduling Algorithm of RC4. In: Vaudenay, S., Youssef, A.M. (eds.) SAC 2001. LNCS, vol. 2259, pp. 1–24. Springer, Heidelberg (2001)
3. Golic, J.: Linear statistical weakness of alleged RC4 keystream generator. In: Fumy, W. (ed.) EUROCRYPT 1997. LNCS, vol. 1233, pp. 226–238. Springer, Heidelberg (1997)
4. Jenkins, R.J.: ISAAC and RC4 (1996), http://burtleburtle.net/bob/rand/isaac.html
5. Klein, A.: Attacks on the RC4 stream cipher (February 27, 2006), http://cage.ugent.be/ klein/RC4/ [last accessed on June 27, 2007]
6. Mantin, I., Shamir, A.: A Practical Attack on Broadcast RC4. In: Matsui, M. (ed.) FSE 2001. LNCS, vol. 2355, pp. 152–164. Springer, Heidelberg (2002)
7. Mantin, I.: A Practical Attack on the Fixed RC4 in the WEP Mode. In: Roy, B. (ed.) ASIACRYPT 2005. LNCS, vol. 3788, pp. 395–411. Springer, Heidelberg (2005)
8. Mantin, I.: Predicting and Distinguishing Attacks on RC4 Keystream Generator. In: Cramer, R.J.F. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 491–506. Springer, Heidelberg (2005)
9. Mantin, I.: Analysis of the stream cipher RC4. Master's Thesis, The Weizmann Institute of Science, Israel (2001)

10. Mironov, I. (Not So) Random Shuffles of RC4. In: Yung, M. (ed.) CRYPTO 2002. LNCS, vol. 2442, pp. 304–319. Springer, Heidelberg (2002)
11. Paul, G., Rathi, S., Maitra, S.: On Non-negligible Bias of the First Output Byte of RC4 towards the First Three Bytes of the Secret Key. In: Proceedings of the International Workshop on Coding and Cryptography, pp. 285–294 (2007)
12. G. Paul and S. Maitra. RC4 State Information at Any Stage Reveals the Secret Key. IACR Eprint Server, eprint.iacr.org, number 2007/208, June 1 (2007); This is an extended version of [13]
13. Paul, G., Maitra, S.: Permutation after RC4 Key Scheduling Reveals the Secret Key. In: Adams, C., Miri, A., Wiener, M. (eds.) SAC 2007. LNCS, vol. 4876, pp. 360–377. Springer, Heidelberg (2007)
14. Paul, G., Maitra, S., Srivastava, R.: On Non-Randomness of the Permutation after RC4 Key Scheduling. In: Boztaş, S., Lu, H.-F(F.) (eds.) AAECC 2007. LNCS, vol. 4851, pp. 100–109. Springer, Heidelberg (2007)
15. Paul, S., Preneel, B.: Analysis of Non-fortuitous Predictive States of the RC4 Keystream Generator. In: Johansson, T., Maitra, S. (eds.) INDOCRYPT 2003. LNCS, vol. 2904, pp. 52–67. Springer, Heidelberg (2003)
16. Paul, S., Preneel, B.: A New Weakness in the RC4 Keystream Generator and an Approach to Improve the Security of the Cipher. In: Roy, B., Meier, W. (eds.) FSE 2004. LNCS, vol. 3017, pp. 245–259. Springer, Heidelberg (2004)
17. A. Roos. A class of weak keys in the RC4 stream cipher. Two posts in sci.crypt, message-id 43u1eh$1j3@hermes.is.co.za and 44ebge$llf@hermes.is.co.za (1995), http://marcel.wanda.ch/Archive/WeakKeys
18. Tews, E.: Email Communications (September 2007)
19. Tews, E., Weinmann, R.P., Pyshkin, A.: Breaking 104 bit WEP in less than 60 seconds. IACR Eprint Server, eprint.iacr.org, number 2007/120, April 1 (2007)
20. Vaudenay, S., Vuagnoux, M.: Passive-only key recovery attacks on RC4. In: Adams, C., Miri, A., Wiener, M. (eds.) SAC 2007. LNCS, vol. 4876. Springer, Heidelberg (2007)
21. Wagner, D.: My RC4 weak keys. Post in sci.crypt, message-id 447o1l$cbj@cnn.Princeton.EDU (September 26, 1995), http://www.cs.berkeley.edu/ daw/my-posts/my-rc4-weak-keys

Appendix A

Lemma 5. $P\big((S_{y+1}[S_{y+1}[y]] = f_y) \wedge (S_{y+1}[y] \leq y)\big) \approx \big(\frac{1}{N} \cdot (\frac{N-1}{N})^{\frac{y(y+1)}{2}}\big) \cdot \big(y(\frac{N-2}{N})^{y-1} + (\frac{N-1}{N})^{y}\big)$, $0 \leq y \leq 31$.

Proof. $S_{y+1}[y] \leq y$ means that it can take $y+1$ many values $0, 1, \ldots, y$. Suppose $S_{y+1}[y] = x$, $0 \leq x \leq y-1$. Then $S_{y+1}[x]$ can equal f_y in the following way.

1. From round 1 (when $i = 0$) through x (when $i = x-1$), j does not touch the indices x and f_y. Thus, after round x, $S_x[x] = x$ and $S_x[f_y] = f_y$. This happens with probability $(\frac{N-2}{N})^x$.
2. In round $x+1$ (when $i = x$), j_{x+1} becomes equal to f_y, and after the swap, $S_{x+1}[x] = f_y$ and $S_{x+1}[f_y] = x$. The probability of this event is $P(j_{x+1} = f_y) = \frac{1}{N}$.

3. From round $x+2$ (when $i = x+1$) through y (when $i = y-1$), again j does not touch the indices x and f_y. This, after round y, $S_y[x] = f_y$ and $S_y[f_y] = x$. This occurs with probability $(\frac{N-2}{N})^{y-x-1}$.
4. In round $y+1$ (when $i = y$), j_{y+1} becomes equal to f_y, and after the swap, $S_{y+1}[y] = S_y[f_y] = x$ and $S_{y+1}[S_{y+1}[y]] = S_{y+1}[x] = S_y[x] = f_y$. This happens with probability $P(j_{y+1} = f_y)$ which is approximately equal to $(\frac{N-1}{N})^{\frac{y(y+1)}{2}}$ for small values of y as in the proof of [12, Lemma 1]. We consider $0 \le y \le 31$ for good approximation.

Considering the above events to be independent, we have
$P\big((S_{y+1}[S_{y+1}[y]] = f_y) \wedge (S_{y+1}[y] = x)\big)$
$= (\frac{N-2}{N})^x \cdot \frac{1}{N} \cdot (\frac{N-2}{N})^{y-x-1} \cdot (\frac{N-1}{N})^{\frac{y(y+1)}{2}} = (\frac{1}{N}) \cdot (\frac{N-2}{N})^{y-1} \cdot (\frac{N-1}{N})^{\frac{y(y+1)}{2}}$.
Summing for all x in $[0, \ldots, y-1]$, we get $P\big((S_{y+1}[S_{y+1}[y]] = f_y) \wedge (S_{y+1}[y] \le y-1)\big) = (\frac{y}{N}) \cdot (\frac{N-2}{N})^{y-1} \cdot (\frac{N-1}{N})^{\frac{y(y+1)}{2}}$.

If $S_{y+1}[y] = y$, then $S_{y+1}[S_{y+1}[y]]$ can equal f_y in the following ways: (a) f_y has to be equal to y; this happens with probability $\frac{1}{N}$, (b) index y is not touched by j in any of the first y rounds; this happens with probability $(\frac{N-1}{N})^y$, and (c) in the $(y+1)$-th round, $j_{y+1} = f_y$ so that there is no swap; this happens with probability $(\frac{N-1}{N})^{\frac{y(y+1)}{2}}$. Hence, $P\big((S_{y+1}[S_{y+1}[y]] = f_y) \wedge (S_{y+1}[y] = y)\big) = (\frac{1}{N}) \cdot (\frac{N-1}{N})^y \cdot (\frac{N-1}{N})^{\frac{y(y+1)}{2}}$.

Adding the above two contributions (one for $0 \le S_{y+1}[y] \le y-1$ and the other for $S_{y+1}[y] = y$), we get $P\big((S_{y+1}[S_{y+1}[y]] = f_y) \wedge (S_{y+1}[y] \le y)\big) = \big(\frac{1}{N} \cdot (\frac{N-1}{N})^{\frac{y(y+1)}{2}}\big) \cdot \big(y(\frac{N-2}{N})^{y-1} + (\frac{N-1}{N})^y\big)$. □

Lemma 6. *Let* $p_r(y) = P\big((S_r[S_r[y]] = f_y) \wedge (S_r[y] \le r-1)\big)$, $0 \le y \le N-1$, $1 \le r \le N$. *Then* $p_r(y) = (\frac{N-2}{N})p_{r-1}(y) + \frac{1}{N} \cdot (\frac{N-y}{N}) \cdot (\frac{N-1}{N})^{\frac{y(y+1)}{2}+2r-3}$, $0 \le y \le 31$, $y+2 \le r \le N$.

Proof. Then event $\big((S_r[S_r[y]] = f_y) \wedge (S_r[y] \le r-1)\big)$, where $r \ge y+2$, can occur in two mutually exclusive and exhaustive ways: $\big((S_r[S_r[y]] = f_y) \wedge (S_r[y] \le r-2)\big)$ and $\big((S_r[S_r[y]] = f_y) \wedge (S_r[y] = r-1)\big)$. We compute the contribution of each separately.

In the r-th round, $i = r-1$ and hence does not touch the indices $0, \ldots, r-2$. Hence the event $\big((S_r[S_r[y]] = f_y) \wedge (S_r[y] \le r-2)\big)$ occurs if we already had $\big((S_{r-1}[S_{r-1}[y]] = f_y) \wedge (S_{r-1}[y] \le r-2)\big)$ and $j_r \notin \{y, S_{r-1}[y]\}$. Thus, the contribution of this part is $p_{r-1}(y) \cdot (\frac{N-2}{N})$.

The event $\big((S_r[S_r[y]] = f_y) \wedge (S_r[y] = r-1)\big)$ occurs if after the $(r-1)$-th round, $S_{r-1}[r-1] = r-1$, $S_{r-1}[y] = f_y$ and in the r-th round (i.e., when $i = r-1$), $j_r = y$ causing a swap involving the indices y and $r-1$.

1. We have $S_{r-1}[r-1] = r-1$ if the location $r-1$ is not touched during the rounds $i = 0, \ldots, r-2$. This happens with probability $(\frac{N-1}{N})^{r-1}$.
2. The event $S_{r-1}[y] = f_y$ happens with a probability which is approximately equal to $(\frac{N-y}{N}) \cdot (\frac{N-1}{N})^{[\frac{y(y+1)}{2}+r-2]}$ for small values of y as in the proof of [12, Theorem 1]. We consider $0 \le y \le 31$ for good approximation.

3. In the r-th round (when $i = r-1$), j_r becomes y with probability $\frac{1}{N}$.

Thus, $P\big((S_r[S_r[y]] = f_y) \wedge (S_r[y] = r-1)\big) = (\frac{N-1}{N})^{r-1} \cdot (\frac{N-y}{N})(\frac{N-1}{N})^{[\frac{y(y+1)}{2}+r-2]} \cdot \frac{1}{N} = \frac{1}{N} \cdot (\frac{N-y}{N}) \cdot (\frac{N-1}{N})^{\frac{y(y+1)}{2}+2r-3}$.

Adding the above two contributions, we get

$p_r(y) = (\frac{N-2}{N})p_{r-1}(y) + \frac{1}{N} \cdot (\frac{N-y}{N}) \cdot (\frac{N-1}{N})^{\frac{y(y+1)}{2}+2r-3}$. □

The recurrence in Lemma 6 and the base case in Lemma 5 completely specify the probabilities $p_r(y)$ for all y in $[0, \ldots, 31]$ and r in $[y+1, \ldots, N]$.

Theorem 2 (Section 2): *After the complete KSA,*
$P(S_N[S_N[y]] = f_y) \approx \frac{y}{N} \cdot (\frac{N-1}{N})^{\frac{y(y+1)}{2}+2(N-2)} + \frac{1}{N} \cdot (\frac{N-1}{N})^{\frac{y(y+1)}{2}-y+2(N-1)} + (\frac{N-y-1}{N}) \cdot (\frac{N-y}{N}) \cdot (\frac{N-1}{N})^{\frac{y(y+1)}{2}+2N-3}$, $0 \leq y \leq 31$.

Proof. Using the approximation $\frac{N-2}{N} \approx (\frac{N-1}{N})^2$, the recurrence in Lemma 6 can be rewritten as $p_r(y) = (\frac{N-1}{N})^2 p_{r-1}(y) + \frac{1}{N}(\frac{N-y}{N}) \cdot (\frac{N-1}{N})^{\frac{y(y+1)}{2}+2r-3}$, i.e., $p_r(y) = ap_{r-1}(y) + a^{r-1}b$, where $a = (\frac{N-1}{N})^2$ and $b = \frac{1}{N}(\frac{N-y}{N}) \cdot (\frac{N-1}{N})^{\frac{y(y+1)}{2}-1}$. The solution of this recurrence is $p_r(y) = a^{r-y-1}p_{y+1}(y) + (r-y-1)a^{r-1}b$, $r \geq y+1$. Substituting the values of $p_{y+1}(y)$ (from Lemma 5), a and b, we get $p_r(y) = \frac{y}{N} \cdot (\frac{N-1}{N})^{\frac{y(y+1)}{2}+2(r-2)} + \frac{1}{N} \cdot (\frac{N-1}{N})^{\frac{y(y+1)}{2}-y+2(r-1)} + (\frac{r-y-1}{N}) \cdot (\frac{N-y}{N}) \cdot (\frac{N-1}{N})^{\frac{y(y+1)}{2}+2r-3}$, $y+1 \leq r \leq N$, for initial values of y $(0 \leq y \leq 31)$. Substituting $r = N$ and noting the fact that $P\big((S_N[S_N[y]] = f_y) \wedge (S_N[y] \leq N-1)\big) = P(S_N[S_N[y]] = f_y)$, we get the result. □

Even after the approximation, our theoretical formula matches closely with the experimental results for $0 \leq y \leq 31$.

Efficient Reconstruction of RC4 Keys from Internal States*

Eli Biham and Yaniv Carmeli

Computer Science Department
Technion – Israel Institute of Technology
Haifa 3200, Israel
{biham,yanivca}@cs.technion.ac.il
http://www.cs.technion.ac.il/∼biham/
http://www.cs.technion.ac.il/∼yanivca/

Abstract. In this paper we present an efficient algorithm for the retrieval of the RC4 secret key, given an internal state. This algorithm is several orders of magnitude faster than previously published algorithms. In the case of a 40-bit key, it takes only about 0.02 seconds to retrieve the key, with success probability of 86.4%. Even if the algorithm cannot retrieve the entire key, it can retrieve partial information about the key. The key can also be retrieved if some of the bytes of the initial permutation are incorrect or missing.

Keywords: Cryptanalysis, Initial Permutation, Key Scheduling, RC4, Stream Cipher.

1 Introduction

The stream cipher RC4 was designed by Ron Rivest, and was first introduced in 1987 as a proprietary software of RSA DSI. The details remained secret until 1994, when they were anonymously published on an internet newsgroup [1]. RSA DSI did not confirm that the published algorithm is in fact the RC4 algorithm, but experimental tests showed that it produces the same outputs as the RC4 software.

More than twenty years after its release, RC4 is still the most widely used software stream cipher in the world. Among other uses, it is used to protect internet traffic as part of the SSL (Secure Socket Layer) and TLS (Transport Layer Security [3]) protocols, and to protect wireless networks as part of the WEP (Wired Equivalent Privacy) and WPA (Wi-Fi Protected Access) protocols.

The state of RC4 consists of a permutation S of the numbers $0, \ldots, N-1$, and two indices $i, j \in \{0, \ldots, N-1\}$, where $N = 256$. RC4 is comprised of two algorithms: the Key Scheduling Algorithm (KSA), which uses the secret key to create a pseudo-random initial state, and the Pseudo Random Generation Algorithm (PRGA), which generates the pseudo-random stream.

[*] This work was supported in part by the Israel MOD Research and Technology Unit.

K. Nyberg (Ed.): FSE 2008, LNCS 5086, pp. 270–288, 2008.

1.1 Previous Attacks

Most attacks on RC4 can be categorized as distinguishing attacks or key-retrieval attacks. Distinguishing attacks try to distinguish between an output stream of RC4 and a random stream, and are usually based on weaknesses of the PRGA. Key recovery attacks recover the secret key, and are usually based on weaknesses of the KSA.

In 1994, immediately after the RC4 algorithm was leaked, Finney [4] showed a class of states that RC4 can never enter. This class consists of states satisfying $j = i+1$ and $S[i+1] = 1$. RC4 preserves the class of Finney states by transferring Finney states to Finney states, and non-Finney states to non-Finney states. Since the initial state (the output of the KSA) is not a Finney state (in the initial state $i = j = 0$) then RC4 can never enter these states. Biham et. al. [2] show how to use Finney states with fault analysis in order to attack RC4.

Knudsen et al. [11] use a backtracking algorithm to mount a known plaintext attack on RC4. They guess the values of the internal state, and simulate the generation process. Whenever the output doesn't agree with the real output, they backtrack and guess another value.

Golić [7] describes a linear statistical weakness of RC4 caused by a positive correlation between the second binary derivative of the least significant bit and 1, and uses it to mount a distinguishing attack.

Fluhrer and McGrew [6] show a correlation between consecutive output bytes, and introduce the notion of k-fortuitous states (classes of states defined by the values of i, j, and only k permutation values, which can predict the outputs of the next k iterations of the PRGA), and build a distinguisher based on that correlation.

Mantin and Shamir generalize the notion of fortuitous states and define b-predictive k-states (states with k known permutation values which predict only b output words, for $b \leq k$) and k-profitable states, which are classes of states in which the index j behaves in the same way for k steps of the PRGA. The predictive states cause certain output sequences to appear more often than expected in a random sequence, thus they are helpful in mounting a distinguishing attack on RC4.

Mantin and Shamir [14] also show that the second word of the output is slightly more probable to be 0 than any other value. Using this bias they are able to build a prefix distinguisher for RC4, based on only about N short streams.

In 2005 Mantin [13] observed that some fortuitous states return to their initial state after the index i leaves them. These states have a chance to remain the same even after a full cycle of N steps, and the same output of the state may be observed again. Mantin uses these states to predict, with high probability, future output bytes of the stream.

In practical applications, stream ciphers are used with a session key which is derived from a shared secret key and an Initial Value (IV, which is transmitted unencrypted). The derivation of the session key can be done in various ways such as concatenating, XORing, or hashing (in WEP, for instance, the secret key is concatenated after the IV).

Many works try to exploit weaknesses of a specific method for deriving the session key. Fluhrer, Mantin, and Shamir [5] have shown a chosen IV attack on the case where the IV precedes the secret key. Using the first output bytes of $60l$ chosen IVs (l is the length of the secret key), they recover the secret key with high probability. They also describe an attack on the case where the IV follows the secret key, which reveals significant information about the internal state just after l steps of the KSA, thus reducing the cost of exhaustive search significantly.

In March 2007, Klein [10] (followed by Tews et. al. [17]) showed a statistical correlation between any output byte and the value of $S[j]$ at the time of the output generation. They use this correlation to retrieve the entire secret key using the first bytes of the output streams of about 40,000 known IVs (for the cases the IV is concatenated either before or after the secret key).

Vaudenay and Vuagnoux [18] improve the attacks of [5,10] on the case of WEP (where the IV is concatenated before the secret key). They present the VX attack, which uses the sum of the the key bytes to reduce the dependency between the other bytes of the key, such that the attack may work even if the data is insufficient to retrieve one of the bytes.

Paul and Maitra [15] use biases in the first entries of the initial permutation to recover the secret key from the initial permutation. They use the first entries of the permutation to create equations which hold with certain probability. They guess some of the bytes of the secret key, and use the equations to retrieve the rest of the bytes. The success of their algorithm relies on the existence of sufficiently many correct equations.

1.2 Outline of Our Contribution

In this paper we present methods that allow us to obtain significantly better results than the algorithm of [15]. A major observation considers the difference between pairs of equations instead of analyzing each equation separately. We show that the probability that the difference of a pair of equations is correct is much higher in most cases than the probabilities of each of the individual equations. Therefore, our algorithm can rely on many more equations and apply more thorough statistical techniques than the algorithm of [15]. We also show two filtering methods that allow us to identify that some of the individual equations (used in [15]) are probably incorrect by a simple comparison, and therefore, to discard these equations and all the differences derived from them. Similarly, we show filtering techniques that discard difference equations, and even correct some of these equations. We also show how to create alternative equations, which can replace the original equations in some of the cases and allow us to receive better statistical information when either the original equations are discarded or they lead to incorrect values. We combine these observations (and other observations that we discuss in this paper) into a statistical algorithm that recovers the key with a much higher success rate than the one of [15]. Our Algorithm also works if some of the bytes of the initial permutation are missing or contain errors. Such scenarios are likely results of side channel attacks, as in [9]. In these cases, our algorithm can even be used to reconstruct the full correct initial permutation by finding the correct key and then using it to compute the correct values. Details of an efficient implementation of the data structures and internals of the algorithm are also discussed.

The algorithm we propose retrieves one linear combination of the key bytes at a time. In each step, the algorithm applies statistical considerations to choose the subset of key bytes participating in the linear combination and the value which have the highest probability to be correct. If this choice turns out to be incorrect, other probable choices may be considered. We propose ways to discover incorrect choices even before the entire key is recovered (i.e., before it can be tested by running the KSA), and thus we are able to save valuable computation time that does not lead to the correct key.

Our algorithm is much faster than the algorithm of [15], and has much better success rates for the same computation time. For example, for 40-bit keys and 86% success rate, our algorithm is about 10000 times faster than the algorithm of [15]. Additionally, even if the algorithm fails to retrieve the full key, it can retrieve partial information about the key. For example, for 128-bit keys it can give a suggestion for the sum of all the key bytes which has a probability of 23.09% to be correct, or give four suggestions such that with a probability of 41.44% the correct value of the sum of all the key bytes is one of the four.

1.3 Organization of the Paper

This paper is organized as follows: Section 2 describes the RC4 algorithms, gives several observation about the keys of RC4, and defines notations which will be used throughout this paper. Section 3 presents the bias of the first bytes of the initial permutation, and describes the attack of [15], which uses these biases to retrieve the secret key. Section 4 gathers several observations which are the building blocks of our key retrieval algorithm, and have enabled us to improve the result of [15]. Section 5 takes these building blocks and uses them together to describe the detailed algorithm. In Section 6 we give some comments and observations about an efficient implementation to our algorithm. Finally, Section 7 summarizes the paper, presents the performance of our algorithm and discusses its advantages over the algorithm of [15].

2 The RC4 Stream Cipher

The internal state of RC4 consists of a permutation S of the numbers $0, \ldots, N-1$, and two indices $i, j \in \{0, \ldots, N-1\}$. The permutation S and the index j form the secret part of the state, while the index i is public and its value at any stage of the stream generation is widely known. In RC4 $N = 256$, and thus the secret internal state has $\log_2 \left(2^8 \cdot 256!\right) \approx 1692$ bits of information. Together with the public value of i there are about 1700 bits of information in the internal state. Variants with other values of N have also been analyzed in the cryptographic literature.

RC4 consists of two algorithms: The Key Scheduling Algorithm (KSA), and the Pseudo Random Generation Algorithm (PRGA), both algorithms are presented in

```
KSA(K)
Initialization:
  For $i = 0$ to $N - 1$
    $S[i] = i$
  $j \leftarrow 0$
Scrambling:
  For $i = 0$ to $N - 1$
    $j \leftarrow j + S[i] + K[i \bmod l]$
    Swap($S[i], S[j]$)
```

```
PRGA(S)
Initialization:
  $i \leftarrow 0$
  $j \leftarrow 0$
Generation loop:
  $i \leftarrow i + 1$
  $j \leftarrow j + S[i]$
  Swap($S[i], S[j]$)
  Output $S[S[i] + S[j]]$
```

Fig. 1. The RC4 Algorithms

Figure 1. All additions in RC4 are performed modulo N. Therefore, in this paper, additions are performed modulo 256, unless explicitly stated otherwise.

The KSA takes an l-byte secret key, K, and generates a pseudo-random initial permutation S. The key size l is bounded by N bytes, but is usually in the range of 5–16 bytes (40–128 bits). The bytes of the secret key are denoted by $K[0], \ldots, K[l-1]$. If $l < N$ the key is repeated to form a N-byte key. The KSA initializes S to be the identity permutation, and then performs N swaps between the elements of S, which are determined by the secret key and the content of S. Note that because i is incremented by one at each step, each element of S is swapped at least once (possibly with itself). On average each element of S is swapped twice.

The PRGA generates the pseudo-random stream, and updates the internal state of the cipher. In each iteration of the PRGA, the values of the indices are updated, two elements of S are swapped, and a byte of output is generated. During the generation of N consecutive output bytes, each element of S is swapped at least once (possibly with itself), and twice on average.

2.1 Properties of RC4 Keys

There are $2^{8 \cdot 256} = 2^{2048}$ possible keys (every key shorter than 256 bytes has an equivalent 256-byte key) but only about 2^{1684} possible initial states of RC4. Therefore, every initial permutation has on average about 2^{364} 256-byte keys which create it. Each initial permutation $\hat{S}$ has at least one, easy to find, 256-byte key: Since every byte of the key is used only once during the KSA, the key bytes are chosen one by one, where $K[i]$ is chosen to set j to be the current location of $\hat{S}[i]$ (which satifies, by this construction $j > i$). Thus, the Swap($S[i]$,$S[j]$) operation on iteration i swaps the value $\hat{S}[i] = S[j]$ with $S[i]$. The value $\hat{S}[i]$ does not participate in later swaps, and thus remains there until the end of the KSA.

The number of initial permutations which can be created by short keys, however, is much smaller. For example, the number of 16-byte keys is only 2^{128}, and the total number of keys bounded by 210 bytes is about $2^{8 \cdot 210} = 2^{1680}$, which is smaller than the total number of permutations.

2.2 Notations

We use the notation $K[a, b]$ to denote the sum of the key bytes $K[a]$ and $K[b]$, i.e.,

$$K[a, b] \triangleq K[a \bmod l] + K[b \bmod l] \bmod N.$$

Similarly, $K[a, b, c]$, $K[a, b, c, d]$, etc., are the sums of the corresponding key bytes for any number of comma-separated arguments. We use the notation $K[a \ldots b]$ to denote the sum of the key bytes in the range $a, a + 1, \ldots, b$, i.e.,

$$K[a \ldots b] \triangleq \sum_{r=a}^{b} K[r \bmod l] \bmod N.$$

We also use combinations of the above, for instance:

$$K[a, b \ldots c] \triangleq K[a \bmod l] + \sum_{r=b}^{c} K[r \bmod l],$$

$$K[a \ldots b, c \ldots d] \triangleq \sum_{r=a}^{b} K[r \bmod l] + \sum_{r=c}^{d} K[r \bmod l].$$

We use the notations S_r and j_r to denote the values of the permutation S and the index j after r iterations of the loop of the KSA have been executed. The initial value of j is $j_0 = 0$ and its value at the end of the KSA is j_N. S_0 is the identity permutation, and S_N is the result of the KSA (i.e., the initial permutation that we study in this paper). For clarity, from now on the notation S (without an index) denotes the initial permutation S_N.

3 Previous Techniques

In 1995 Roos [16] noticed that some of the bytes of the initial permutation have a bias towards a linear combination of the secret key bytes. Theorem 1 describes this bias (the theorem is taken from [16], but is adapted to our notations).

Theorem 1. *The most likely value for $S[i]$ at the end of the KSA is:*

$$S[i] = K[0 \ldots i] + \frac{i(i+1)}{2} \quad \bmod N. \tag{1}$$

Only experimental results for the probabilities of the biases in Theorem 1 are provided in [16]. Recently, Paul and Maitra [15] supplied an analytic formula for this probability, which has corroborated the results given by [16]. Theorem 2 presents their result.

Theorem 2 (Corollary 2 of [15]). *Assume that during the KSA the index j takes its values uniformly at random from $\{0, 1, \ldots, N-1\}$. Then,*

$$P\left(S[i] = K[0 \ldots i] + \frac{i(i+1)}{2}\right) \geq \left(\frac{N-i}{N}\right) \cdot \left(\frac{N-1}{N}\right)^{\frac{i(i+1)}{2}+N} + \frac{1}{N}.$$

For any fixed value of i, the bias described by (1) is the result of a combination of three events that occur with high probability:

1. $S_r[r] = r$ for $r \in \{0, \ldots, i\}$ (i.e., the value of $S[r]$ was not swapped before the r-th iteration).
2. $S_i[j_{i+1}] = j_{i+1}$.
3. $j_r \neq i$ for $r \in \{i+1, \ldots, N-1\}$.

If the first event occurs then the value j_{i+1} is affected only by the key bytes and constant values:

$$j_{i+1} = \sum_{r=0}^{i} (K[r] + S_r[r]) = \sum_{r=0}^{i} (K[r] + r) = K[0 \ldots i] + \frac{i\,(i+1)}{2}.$$

If the second event occurs, then after $i+1$ iteration of the KSA $S_{i+1}[i] = j_{i+1}$. The third event ensures that the index j does not point to $S[i]$ again, and therefore $S[i]$ is not swapped again in later iterations of the KSA. If all three events occur then (1) holds since

$$S_N[i] \underset{\overset{\uparrow}{3}}{=} S_{i+1}[i] \underset{\overset{\uparrow}{2}}{=} j_{i+1} = \sum_{r=0}^{i} (K[r] + S_r[r]) \underset{\overset{\uparrow}{1}}{=} K[0 \ldots i] + \frac{i\,(i+1)}{2}.$$

The probabilities derived from Theorem 2 for the biases of the first 48 entries of S ($S[0] \ldots S[47]$) are given in Table 1 (also taken from [15]). It can be seen that this probability is about 0.371 for $i = 0$, and it decreases as the value of i increases. For $i = 47$ this probability is only 0.008, and for further entries it becomes too low to be used by the algorithm (the a-priori probability that an entry equals any random value is $1/256 \approx 0.0039$). The cause for such a decrease in the bias is that the first of the aforementioned events is less likely to occur for high values of i, as there are more constraints on entries in S.

Given an initial permutation S (the result of the KSA), each of its entries can be used to derive a linear equation of the key bytes, which holds with the probability given by Theorem 2. Let C_i be defined as

$$C_i = S[i] - \frac{i \cdot (i+1)}{2}.$$

Table 1. The Probabilities Given by Theorem 2

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Prob.	.371	.368	.364	.358	.351	.343	.334	.324	.313	.301	.288	.275	.262	.248	.234	.220
i	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Prob.	.206	.192	.179	.165	.153	.140	.129	.117	.107	.097	.087	.079	.071	.063	.056	.050
i	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
Prob.	.045	.039	.035	.031	.027	.024	.021	.019	.016	.015	.013	.011	.010	.009	.008	.008

Using (1) the i'th equation (derived from the entry $S[i]$) becomes:

$$K[0 \ldots i] = C_i. \tag{2}$$

The RecoverKey algorithm of [15] uses these equations in order to retrieve the secret key of RC4. Let n and m be parameters of the algorithm, and recall that l is the length of the secret key in bytes. For each combination of m independent equations out of the first n equations of (2), the algorithm exhaustively guesses the value of $l - m$ key bytes, and solves the m equations to retrieve the rest of the key bytes. The success of the RecoverKey algorithm relies on the existence of m correct and linearly independent equations among the first n equations. The success probabilities and the running time of the RecoverKey algorithm for different key sizes and parameters, as given by [15], are presented in Table 2.[1]

4 Our Observations

Several important observations allow us to suggest an improved algorithm for retrieving the key from the initial permutation.

4.1 Subtracting Equations

Let $i_2 > i_1$. As we expect that $K[0 \ldots i_1] = C_{i_1}$ and $K[0 \ldots i_2] = C_{i_2}$, we also expect that

$$K[0 \ldots i_2] - K[0 \ldots i_1] = K[i_1 + 1 \ldots i_2] = C_{i_2} - C_{i_1} \tag{3}$$

holds with the product of the probabilities of the two separate equations. However, we observe that this probability is in fact much higher. If the following three events occur then (3) holds (compare with the three events described in Section 3):

1. $S_r[r] = r$ for $r \in \{i_1 + 1, \ldots, i_2\}$ (i.e., the value of $S[r]$ was not swapped before the r-th iteration).
2. $S_{i_1}[j_{i_1+1}] = j_{i_1+1}$ and $S_{i_2}[j_{i_2+1}] = j_{i_2+1}$.
3. $j_r \neq i_1$ for $r \in \{i_1 + 1, \ldots, N - 1\}$, and $j_r \neq i_2$ for $r \in \{i_2 + 1, \ldots, N - 1\}$.

[1] We observe that the formula for the complexity given in [15] is mistaken, and the actual values should be considerably higher than the ones cited in Table 2. We expect that the correct values are between 2^5 and 2^8 times higher. The source for the mistake is two-fold: the KSA is considered as taking one unit of time, and the complexity analysis is based on an inefficient implementation of their algorithm. Given a set of l equations, their implementation solves the set of equations separately for every guess of the remaining $l - m$ variables, while a more efficient implementation would solve them only once, and only then guess the values of the remaining bytes. Our complexities are even lower than the complexities given in [15], and are much lower than the correct complexities.

We also observe that the complexities given by [15] for the case of 16-byte keys do not match the formula they publish (marked by * in Table 2). The values according to their formula should be 2^{82}, 2^{79}, 2^{73} and 2^{69} rather than 2^{60}, 2^{63}, 2^{64} and 2^{64}, respectively. Their mistake is possibly due to an overflow in 64-bit variables.

Table 2. Success Probabilities and Running Time of the RecoverKey Algorithm of [15]

l	n	m	Time	$P_{Success}$
5	16	5	2^{18}	0.250
5	24	5	2^{21}	0.385
8	16	6	2^{34}	0.273
8	20	7	2^{29}	0.158
8	40	8	2^{33}	0.092
10	16	7	2^{43}	0.166
10	24	8	2^{40}	0.162
10	48	9	2^{43}	0.107
12	24	8	2^{58}	0.241
12	24	9	2^{50}	0.116
16	24	9	2^{60} *	0.185
16	32	10	2^{63} *	0.160
16	32	11	2^{64} *	0.086
16	40	12	2^{64} *	0.050

* Incorrect entries — see footnote [1].

If the first event occurs then the index j is affected in iterations i_1+1 through i_2 only by the key bytes and constant values:

$$j_{i_2+1} - j_{i_1+1} = \sum_{r=i_1+1}^{i_2} \left(K[r] + S_r[r]\right) = K[i_1+1\ldots i_2] + \sum_{r=i_1+1}^{i_2} r$$

If the second event occurs, then after i_1+1 iteration of the KSA $S_{i_1+1}[i_1] = j_{i_1+1}$, and after i_2+1 iteration $S_{i_2+1}[i_2] = j_{i_2+1}$. The third event ensures that the index j does not point to $S[i_1]$ or $S[i_2]$ again, and therefore $S[i_1]$ and $S[i_2]$ are not swapped again in later iterations. If all three events occur then (3) holds since

$$\begin{aligned} S_N[i_2] - S_N[i_1] &\underset{\overset{\uparrow}{3}}{=} S_{i_2+1}[i_2] - S_{i_1+1}[i_1] \underset{\overset{\uparrow}{2}}{=} j_{i_2+1} - j_{i_1+1} = \\ &= \sum_{r=i_1+1}^{i_2} \left(K[r] + S_r[r]\right) \underset{\overset{\uparrow}{1}}{=} K[i_1+1\ldots i_2] + \sum_{r=i_1+1}^{i_2} r = \\ &= K[i_1+1\ldots i_2] + \frac{i_2\left(i_2+1\right)}{2} - \frac{i_1\left(i_1+1\right)}{2}, \end{aligned}$$

and therefore

$$K[i_1+1\ldots i_2] = C_{i_2} - C_{i_1}\ .$$

Theorem 3 states the exact bias of such differences.

Theorem 3. *Assume that during the KSA the index j takes its values uniformly at random from $\{0,1,\ldots,N-1\}$, and let $0 \le i_1 < i_2 < N$. Then,* $P\left(C_{i_2} - C_{i_1} = K[i_1+1\ldots i_2]\right) \ge$

$$\left[\left(1-\tfrac{i_2}{N}\right)^2 \cdot \left(1-\tfrac{i_2-i_1+2}{N}\right)^{i_1} \cdot \left(1-\tfrac{2}{N}\right)^{N-i_2-1} \cdot \prod_{r=0}^{i_1-i_2-1}\left(1-\tfrac{r+2}{N}\right)\right] + \tfrac{1}{N}.$$

The proof of Theorem 3 is based on the discussion which precedes it, and is similar to the proof of Theorem 2 given in [15]. The proof is based on the analysis of the probabilities that the values of j throughout the KSA are such that the three events described earlier hold.

As a result of Theorem 3 our algorithm has many more equations to rely on. We are able to use the difference equations which have high enough probability, and furthermore, we can now use data which was unusable by the algorithm of [15]. For instance, according to Theorem 2, the probability that $K[0 \ldots 50] = C_{50}$ is 0.0059, and the probability that $K[0 \ldots 52] = C_{52}$ is 0.0052. Both equations are practically useless by themselves, but according to Theorem 3 the probability that $K[51 \ldots 52] = C_{52} - C_{50}$ is 0.0624, which is more than ten times the probabilities of the individual equations.

Moreover, the biases given by Theorem 2 and used by the RecoverKey algorithm of [15] are dependent. If $S_r[r] \neq r$ for some r, then the first event described in Section 3 is not expected to hold for any $i > r$, and we do not expect equations of the form (2) to hold for such values of i. However, equations of the form (3) for $i_2 > i_1 > r$ may still hold under these conditions, allowing us to handle initial permutations which the RecoverKey algorithm cannot.

4.2 Using Counting Methods

Since every byte of the secret key is used more than once, we can obtain several equations for the same sum of key bytes. For example, all the following equations:

$$\begin{aligned} C_1 &= K[0 \ldots 1] \\ C_{l+1} - C_{l-1} &= K[l \ldots l+1] = K[0 \ldots 1] \\ C_{2l+1} - C_{2l-1} &= K[2l \ldots 2l+1] = K[0 \ldots 1] \end{aligned}$$

suggest values for $K[0] + K[1]$. If we have sufficiently many suggestions for the same sum of key bytes, the correct value of the sum is expected to appear more frequently than other values. We can assign a weight to each suggestion, use counters to sum the weights for each possible candidate, and select the candidate which has the highest weight. We may assign the same weight to all the suggestions (majority vote) or a different weight for each suggestion (e.g., according to its probability to hold, as given by Theorems 2 and 3). We demonstrate the use of counters using the previous example. Assume that $C_1 = 178$, $C_{l+1} - C_{l-1} = 210$ and $C_{2l+1} - C_{2l-1} = 178$ are the only suggestions for the value of $K[0 \ldots 1]$, and assume that all three suggestions have equal weights of one. Under these conditions the value of the counter of 178 will be two, the value of the counter of 210 will be one, and all other counters will have a value of zero. We guess that $K[0 \ldots 1] = 178$, since it has the highest total weight of suggestions.

A simple algorithm to retrieve the full key would be to look at all the suggestions for each of the key bytes, and choose the most frequent value for each one. Unfortunately, some of the bytes retrieved by this sort of algorithm are expected to be incorrect. We can run the KSA with the retrieved key to test its correctness, but if the test fails (the KSA does not produce the expected initial permutation), we get no clue to where the mistake is.

However, we observe that we do not need to limit ourselves to a single key byte, but rather consider all candidates for all possible sums of key bytes suggested by the equations, and select the combination with the highest total weight. Once we fix the chosen value for the first sum, we can continue to another, ordered by the weight, until we have the entire key. There is no need to consider sequences which are linearly dependent in prior sums. For example, if we have already fixed the values of $K[0] + K[1]$ and $K[0]$, there is no need to consider suggestions for $K[1]$. Therefore, we need to set the values of exactly l sums in order to retrieve the full key. Moreover, each value we select allows us to substantially reduce the number of sums we need to consider for the next step, as it allows us to merge the counters of some of the sums (for example, if we know that $K[0]+K[1] = 50$ then we can treat suggestions for $K[0] = 20$ together with $K[1] = 30$).

A natural extension to this approach is trying also the value with the second highest counter, in cases where the highest counter is found wrong. More generally, once a value is found wrong, or a selection of a sequence is found unsatisfactory, backtracking is performed. We denote the number of attempts to be tried on the t-th guess by λ_t, for $0 \leq t < l$. This method can be thought of as using a DFS algorithm to search an ordered tree of height $l + 1$, where the degree of vertices on the t-th level is λ_t and every leaf represents a key.

4.3 The Sum of the Key Bytes

Denote the sum of all l key bytes by s, i.e.,

$$s = K[0 \ldots l - 1] = \sum_{r=0}^{l-1} K[r].$$

The value of s is especially useful. The linear equations derived from the initial permutation give sums of sequences of consecutive key bytes. If we know the value of s, all the suggestions for sequences longer than l bytes can be reduced to suggestions for sequences which are shorter than l bytes. For example, from the following equations:

$$\begin{aligned} C_1 &= K[0 \ldots 1] \\ C_{l+1} &= K[0 \ldots l+1] = s + K[0 \ldots 1] \\ C_{2l+1} &= K[0 \ldots 2l+1] = 2s + K[0 \ldots 1] \end{aligned}$$

we get three suggestions C_1, $C_{l+1} - s$, and $C_{2l+1} - 2s$ for the value of $K[0]+K[1]$.

After such a reduction is performed, all the remaining suggestions reduce to sums of fewer than l key bytes, of the form $K[i_1 \ldots i_2]$, where $0 \leq i_1 < l$ and $i_1 \leq i_2 < i_1+l-1$. Thus, there are only $l\cdot(l-1)$ possible sequences of key bytes to consider. Furthermore, the knowledge of s allows us to unify every two sequences which sum up to $K[0 \ldots l-1] = s$ (as described in Section 4.2), thus reducing the number of sequences to consider to only $l\cdot(l-1)/2$ (without loss of generality, the last byte of the key, $K[l-1]$, does not appear in the sequences we consider, so each sum we consider is of the form $K[i_1 \ldots i_2]$, for $0 \leq i_1 \leq i_2 < l-1$). In turn, there are more suggestions for each of those unified sequences than there were for each original sequence.

Table 3. Probabilities that s is Among the Four Highest Counters

Key Length	Highest Counter	Second Highest	Third Highest	Fourth Highest
5	0.8022	0.0618	0.0324	0.0195
8	0.5428	0.1373	0.0572	0.0325
10	0.4179	0.1604	0.0550	0.0332
12	0.3335	0.1618	0.0486	0.0287
16	0.2309	0.1224	0.0371	0.0240

Fortunately, besides being the most important key byte sequence, s is also the easiest sequence to retrieve, as it has the largest number of suggestions. Any sum of l consecutive bytes, of the form $K[i+1\ldots i+l] = C_{i+l} - C_i$, for any i, yields a suggestion for s. In a similar way, we can consider sequences of $2l$ bytes for suggestions for $2s$, and we can continue to consider sequences of αl consecutive bytes, for any integer α. However, for common key lengths, the probability of a correct sum with $\alpha > 2$ is too low.

As discussed in Section 4.2, we may want to consider also the second highest counter and perform backtracking. Our experimental results for the success probabilities of retrieving s are presented in Table 3. For each of the key lengths in the table, we give the probability that the value of s is the value with the highest counter, second highest, third highest, or fourth highest. The data in the table was compiled by testing 1,000,000,000 random keys for each of the key lengths, and considering all suggestions with a probability higher than 0.01.

4.4 Adjusting Weights and Correcting Equations

During the run of the algorithm, we can improve the accuracy of our guesses based on previous guesses. Looking at all suggestions for sequences we have already established, we can identify exactly which of them are correct and which are not, and use this knowledge to gain information about intermediate values of j and S during the execution of the KSA. We assume that if a suggestion $C_{i_2} - C_{i_1}$ for $K[i_1+1\ldots i_2]$ is correct, then all three events described in Section 4.1 occur with a relatively high probability. Namely, we assume that:

- $S_r[r] = r$ for $i_1 + 1 \leq r \leq i_2$ (follows from event 1 from Section 4.1).
- $S[i_1] = j_{i_1+1}$ and $S[i_2] = j_{i_2+1}$ (together, follow from events 2 and 3 from Section 4.1.

This information can be used to better assess the probabilities of other suggestions. When considering a suggestion $C_{i_4} - C_{i_3}$ for a sum of key bytes $K[i_3+1\ldots i_4]$ which is still unknown, if we have an indication that one of the three events described in Section 4.1 is more likely to have occurred than predicted by its a-priori probability, the weight associated with the suggestion can be increased. Example 1 demonstrates a case in which such information is helpful.

Example 1. Assume that the following three suggestions are correct:

1. $K[0 \ldots 9] = C_9$,
2. $K[12 \ldots 16] = C_{16} - C_{11}$,
3. $K[7 \ldots 14] = C_{14} - C_6$,

and assume that for each of them the three events described in Section 4.1 hold during the execution of the KSA. From the first suggestion we conclude that $j_{10} = S[9]$, from the second suggestions we learn that $j_{12} = S[11]$, and the third suggestion teaches us that $S_r[r] = r$ for $7 \leq r \leq 14$ (and in particular for r=10 and r=11). It can be inferred from the last three observations and according to the explanation in Section 4.1 that under these assumptions $K[10 \ldots 11] = C_{11} - C_9$. Since the probabilities that the assumptions related to $K[10 \ldots 11] = C_{11} - C_9$ hold are larger than the a-priory probability (due to the relation to the other suggestions, which are known to be correct), the probability that this suggestion for $K[10 \ldots 11]$ is correct is increased.

Similarly, we can gain further information from the knowledge that suggestions are incorrect. Consider r's for which there are many incorrect suggestions that involve C_r, either with preceding C_{i_1} ($C_r - C_{i_1}$, $i_1 < r$) or with succeeding C_{i_2} ($C_{i_2} - C_r$, $i_2 > r$). In such cases we may assume that $S_N[r]$ is not the correct value of j_{r+1}, and thus all other suggestions involving C_r are also incorrect.

Consider r's for which there are many incorrect suggestions that pass over r, i.e., of the form $C_{i_2} - C_{i_1}$ where $i_1 < r \leq i_2$. In this case, we may assume that during the KSA $S_r[r] \neq r$, and thus all other suggestions that pass over r are also incorrect. All suggestions that pass over r for which $S_r[r] \neq r$ is the only event (of the three events described in Section 4.1) that does not hold, must have the same error $\Delta = C_{i_2} - C_{i_1} - K[i_1 + 1 \ldots i_2]$ (which is expected to be $\Delta = S_r[r] - r$). Thus, if we find that for some r several suggestions that pass over r have the same error Δ, we can correct other suggestions by Δ.

4.5 Refining the Set of Equations

We observe that some of the equations can be discarded based on the values of the initial permutation, and some others have alternatives. This observation is also applicable to the equations used by the algorithm of [15], and could have improved its running time and success probabilities.

If $S[i'] < i'$ for some i', then the equation derived from $S[i']$ should be discarded, since $x = S[i']$ is not expected to satisfy (1). In this case, even if Event 1 and Event 3 (of the three events described in Section 3) hold, it is clear that Event 2 does not, as the number x has already been swapped in a previous iteration (when $i = x$), and is not likely to be in location $S[i']$ after i' iterations of the KSA.

If $S[i'] > i'$ for some i', then an alternative equation may be derived from $x = S[i']$, in addition to the equation derived by the algorithm of [15]. The equations used by [15] assume that the assignment $S[i'] \leftarrow x$ occurred with $i = i'$, and $j = S[j] = x$ (Figure 2(a)). However, in this case, another likely possibility is that the assignment $S[i'] \leftarrow x$ occurred with $i = S[i] = x$, and

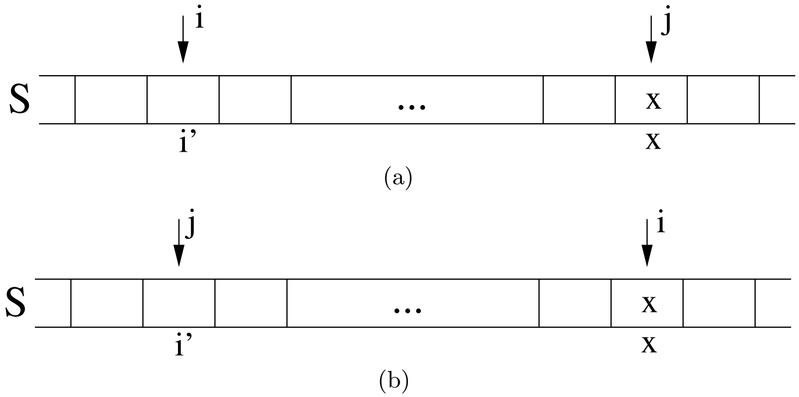


Fig. 2. Two Probable Alternatives to the Positions of the Indices i and j Right Before the Assignment $S[i'] \leftarrow x$ Occurred

$j = i'$ (Figure 2(b)). In the latter case, $j_{x+1} = i'$, and the following equation holds with a high probability:

$$i' = K[0 \ldots x] + \frac{x(x+1)}{2}.$$

It can be shown that this equation holds with a probability slightly higher than the probability given by Theorem 2 for $i = x$. We now have two likely possibilities for the value of j_{x+1}, i' and $S[x]$, which yield two alternative equations. Let $\bar{C}_x$ be defined as:

$$\bar{C}_x = S^{-1}[x] - \frac{x(x+1)}{2} \ .$$

Using this notation, the proposed alternative equation is

$$K[0 \ldots x] = \bar{C}_x \ .$$

Every time C_x is used to create a suggestion (by subtracting equations), the value $\bar{C}_x$ (if exists) can replace it to create an alternative suggestion for the same sum of key bytes. It can be shown that the probabilities that $\bar{C}_{x_2} - C_{x_1}$, $C_{x_2} - \bar{C}_{x_1}$ and $\bar{C}_{x_2} - \bar{C}_{x_1}$ hold are slightly higher than the probability that $C_{x_2} - C_{x_1}$ holds (for any $x_1 < x_2$). Note that we do not expect that many equations have such alternatives, because under the assumption that j takes its values uniformly at random, it is much more likely that $j_{i+1} > i$ for small values of i. Given the two alternatives it is possible to run the algorithm twice, while on each run consider only suggestions derived from the set of equations with one of the two alternatives. However, due to our use of counting methods, both equations can be added to the same set of equations, such that suggestions derived from both alternatives are counted together, in the same counting process.

4.6 Heuristic Pruning of the Search

In Section 4.2 we have described the backtracking approach to finding the key as a DFS search on an ordered tree. Once a guessed value is found wrong (the

```
FIND_KEY($S$)
1. Build the equations: Compute the values of $\{C_i\}$ and $\{\bar{C}_i\}$, for the indices $i$
   where they exist (described in Sections 3 and 4.5).
2. Sum the weights of suggestions for each of the $N$ candidates for $s$ (described
   in Section 4.3).
3. For $x = 1$ to $\lambda_0$ do:
   (a) Find a candidate for $s$ with the highest counter, $w_0$, which has not been
       checked yet, and set $s = K[0 \ldots l-1] = w_0$.
   (b) Mark the correct suggestions for $s = w_0$, adjust weights and correct
       remaining suggestions accordingly (described in Section 4.4).
   (c) Initialize $N$ counters for each sequence of key bytes $K[i_1 \ldots i_2]$ such that
       $0 \leq i_1 \leq i_2 < l-1$, and sum the weights of suggestions for each of them
       (described in Sections 4.1, 4.2 and 4.3).
   (d) Call REC_SUBROUTINE(1) to retrieve the rest of the key. If the correct
       key is found, return it.
4. Return FAIL.
```

Fig. 3. The FIND_KEY Algorithm

keys obtained from it fail to create the requested permutation) we go back and try the other likely guesses. Naturally, by trying more guesses we increase our chances to successfully retrieve the key, but we increase the computation time as well. If we can identify an internal nodes as representing a wrong guess, we can skip the search of the entire subtree rooted from it, and thus reduce the computation time.

Section 4.2 also describes the merging of counters of different sequences according to previous guesses, which allows us to consider fewer key sequences, with more suggestions for each. If the guesses that we have already made are correct, we expect that after such a merge the value of the highest counter is significantly higher than other counters. If the former guesses are incorrect, we don't expect to observe such behavior, as the counters of different sequences will be merged in a wrong way.

Let μ_t for $0 \leq t < l$ be a threshold for the t-th guess. When considering candidates for the t-th guess, we only consider the ones with a counter value of at least μ_t. The optimal values of the thresholds can be obtained empirically, and depend on the key length (l), the weights given to the suggestions, and the number of attempts for each guess (λ_t's). Even if the use of these thresholds may cause correct guesses to be aborted, the overall success probability may still increase, since the saved computation time can be used to test more guesses.

5 The Algorithm

The cornerstones of our method have been laid in Section 4. In this section we gather all our previous observations, and formulate them as an algorithm to retrieve the secret key from the initial permutation S. The FIND_KEY algorithm (presented in Figure 3) starts the search by finding s, and calls the recursive algorithm REC_SUBROUTINE (Figure 4). Each recursive call guesses another value for a sum of key bytes, as described in the previous section.

The optimal values of the parameters $\lambda_0, \ldots, \lambda_{l-1}, \mu_1, \ldots, \mu_{l-1}$ used by the algorithm and the weights it assigns to the different suggestions can by empirically found, so that the success probability of the algorithm and/or the average running time are within a desired range.

6 Efficient Implementation

Recall that on each iteration of the algorithm some of the sums of the key bytes are already known (or guessed). The suggestions for the unknown sums are counted using a set of N counters, one counter for each possible value of that sum. In Section 4.2 we stated that according to the prior guesses, the suggestions for several sums of key bytes may be counted together (i.e., after a new guess is made, some of the counters may be merged with counters of other sums). This section describes an efficient way to discover which counters should be merged, and how to merge them.

The known bytes induce an equivalence relation between the unknown sums of the key bytes. Two sums are in the same equivalence class if and only if the value of each of them can be computed from the value of the other and the values of known sums. We only need to keep a set of N counters for each equivalence class, as all

```
REC_SUBROUTINE($t$)

1. If $t = l$, extract the key from all the $l$ guesses made so far, and verify it. If the key is correct, return it. Otherwise, return FAIL.
2. For $y = 1$ to $\lambda_t$ do:
    (a) Find a combination of key sequence and a candidate for its sum, with the highest counter among the sum of sequences that hasn't already been guessed yet. Denote them by $K[i_1 \ldots i_2]$ and $w_t$, respectively, and denote the value of that counter by $h$.
    (b) If $h < \mu_t$, return FAIL (described in Section 4.6).
    (c) Set $K[i_1 \ldots i_2] = w_t$.
    (d) Mark the correct suggestions for $K[i_1 \ldots i_2] = w_t$, adjust weights and correct remaining suggestions accordingly (described in Section 4.4).
    (e) Merge the counters which may be unified as a result of the guess from 2a (described in Section 4.2).
    (f) Call REC_SUBROUTINE($t + 1$). If the correct key is found, return it. Otherwise, cancel the most recent guess (revert any changes made during the current iteration, including the merging of the counters).
3. Return FAIL.
```

Fig. 4. The Recursive REC_SUBROUTINE Algorithm

suggestions for sums which are in the same equivalence class should be counted together. When we merge counters, we actually merge equivalence classes.

We represent our knowledge about the values of the sums as linearly independent equations of the key bytes. After r key sums are guessed, there are r linear equations of the form

$$\sum_{i=0}^{l-1} a_{i,j} K[j] = b_j,$$

for $1 \leq j \leq r$, where $0 \leq a_{i,j} < N$. The equations are represented as a triangular system of equations, in which the leading coefficient (the first non-zero coefficient) of each equation is one. These r equations form a basis of a linear subspace of all the sums we already know. In this representation the equivalence class of any sum of key bytes $K[i_1 \ldots i_2]$ can be found efficiently: We represent the sum as a linear equation of the key bytes, and apply the Gaussian elimination process, such that the system of equations is kept triangular, and the leading coefficient of each equation is one. Sums from the same equivalence class give the same result, as they all extend the space spanned by the r equations to *the same* larger space spanned by $r+1$ equations. The resulting unique equation can be used as an identifier of the equivalence class. When the counters are merged after a guess of a new value, the same process is applied — we apply Gaussian elimination to the equation representing the current equivalence class in order to discover the equivalence class it belongs to on the next level, and merge the counters. Note that as a result of the Gaussian elimination process we also learn the exact linear mapping between the counters of the current equivalence classes, and the counters of the classes of the next step.

7 Discussion

In this paper we presented an efficient algorithm for the recovery of the secret key from the initial state of RC4, using the first bytes of the permutation. The presented algorithm can also work if only some of the bytes of the initial permutation are known. In this case, suggestions are derived only from the known bytes, and the algorithm is only able to retrieve values of sums of key bytes for which suggestions exist. However, as a result of the reduced number of suggestions the success rates are expected to be lower. The algorithm can also work if some of the bytes contain errors, as the correct values of the sums of key bytes are still expected to appear more frequently than others.

Since changes to the internal state during the stream generation (PRGA) are reversible, our algorithm can also be applied given an internal state at any point of the stream generation phase. Like in [15], our algorithm is also applicable given an intermediate state during the KSA, i.e., S_i $(i < N)$, instead of S_N.

We tested the running times and success probabilities of our algorithm for different key lengths, as summarized in Table 4. The tests were performed on a Pentium IV 3GHz CPU. The running times presented in the table are averaged over 10000 random keys. We have assigned a weight of two to suggestions with

Table 4. Empirical Results of The Proposed Attack

Key Length	Time	$P_{Success}$	Time of Improved [15]* $_{[sec]}$
5	0.02	0.8640	366
8	0.60	0.4058	2900
10	1.46	0.0786	183
10	3.93	0.1290	2932
12	3.04	0.0124	100
12	7.43	0.0212	1000
16	278	0.0005	500

* Our rough estimation for the time it would take an improved version of the algorithm of [15] achieve the same $P_{Success}$ (see footnote [1]). The time of the algorithm of [15] is much slower.

probability higher than 0.05, a weight of one to suggestions with probability between 0.008 and 0.05 and a weight of zero to all other suggestions. The values of the parameters $\lambda_0, \ldots, \lambda_{l-1}, \mu_1, \ldots, \mu_{l-1}$ were chosen in an attempt to achieve the best possible success probability with a reasonable running time. As can be seen in the table, our algorithm is much faster than the one of [15] for the same success rate, and in particular in the case of 5-byte keys, it is about 10000 times faster. Note that with the same computation time, our algorithm achieves about four times the success rate compared to [15] in most presented cases.

Another important advantage of our algorithm over the algorithm of [15] is that when the algorithm of [15] fails to retrieve the key, there is no way to know which of the equations are correct, nor is it possible to retrieve partial information about the key. However, in our algorithm, even if the algorithm fails to retrieve the full key, its first guesses are still likely to be correct, as those guesses are made based on counters with high values. This difference can be exemplified by comparing the success rates of obtaining the sum of key bytes s (Table 3) with the success rates of obtaining the entire key (Table 4).

Acknowledgments

The authors would like to thank Adi Shamir for his comments.

References

1. Anonymous, RC4 Source Code, CypherPunks mailing list, September 9 (1994), http://cypherpunks.venona.com/date/1994/09/msg00304.html
2. Biham, E., Granboulan, L., Nguyễn, P.Q.: Impossible Fault Analysis of RC4 and Differential Fault Analysis of RC4. In: Gilbert, H., Handschuh, H. (eds.) FSE 2005. LNCS, vol. 3557, pp. 359–367. Springer, Heidelberg (2005)

3. Dierks, T., Allen, C.: The TLS Protocol, Version 1.0, Internet Engineering Task Force (January 1999), ftp://ftp.isi.edu/in-notes/rfc2246.txt
4. Finney, H.: An RC4 Cycle That Can't Happen, Usenet newsgroup sci.crypt (September 1994)
5. Fluhrer, S., Mantin, I., Shamir, A.: Weaknesses in the Key Scheduling Algorithm of RC4. In: Vaudenay, S., Youssef, A.M. (eds.) SAC 2001. LNCS, vol. 2259, pp. 1–24. Springer, Heidelberg (2001)
6. Fluhrer, S.R., McGrew, D.A.: Statistical Analysis of the Alleged RC4 Keystream Generator. In: Schneier, B. (ed.) FSE 2000. LNCS, vol. 1978, pp. 19–30. Springer, Heidelberg (2001)
7. Golić, J.D.: Linear Statistical Weakness of Alleged RC4 Keystream Generator. In: Fumy, W. (ed.) EUROCRYPT 1997. LNCS, vol. 1233, pp. 226–238. Springer, Heidelberg (1997)
8. Grosul, A.L., Wallach, D.S.: A Related-Key Cryptanalysis of RC4, Technical Report TR-00-358, Department of Computer Science, Rice University (June 2000), http://cohesion.rice.edu/engineering/computerscience/tr/TR_Download.cfm?SDID=126
9. Halderman, J.A., Schoen, S.D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J.A., Feldman, A.J., Appelbaum, J., Felten, E.W.: Lest We Remember: Cold Boot Attacks on Encryption Keys (February 2008), http://citp.princeton.edu/pub/coldboot.pdf
10. Klein, A.: Attacks on the RC4 Stream Cipher (2007), http://cage.ugent.be/~klein/RC4/RC4-en.ps
11. Knudsen, L.R., Meier, W., Preneel, B., Rijmen, V., Verdoolaege, S.: Analysis Methods for (Alleged) RC4. In: Ohta, K., Pei, D. (eds.) ASIACRYPT 1998. LNCS, vol. 1514, pp. 327–341. Springer, Heidelberg (1998)
12. Mantin, I.: Analysis of the Stream Cipher RC4, Master Thesis, The Weizmann Institute of Science, Israel (2001), http://www.wisdom.weizmann.ac.il/~itsik/RC4/Papers/Mantin1.zip
13. Mantin, I.: Predicting and Distinguishing Attacks on RC4 Keystream Generator. In: Cramer, R.J.F. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 491–506. Springer, Heidelberg (2005)
14. Mantin, I., Shamir, A.: A Practical Attack on Broadcast RC4. In: Matsui, M. (ed.) FSE 2001. LNCS, vol. 2355, pp. 152–164. Springer, Heidelberg (2002)
15. Paul, G., Maitra, S.: Permutation After RC4 Key Scheduling Reveals the Secret Key. In: Adams, C., Miri, A., Wiener, M. (eds.) LNCS, vol. 4876, pp. 260–377. Springer, Heidelberg (2007), http://eprint.iacr.org/2007/208.pdf
16. Roos, A.: A Class of Weak Keys in the RC4 Stream Cipher, Two posts in sci.crypt (1995), http://marcel.wanda.ch/Archive/WeakKeys
17. Tews, E., Weinmann, R.P., Pyshkin, A.: Breaking 104 Bit WEP in Less than 60 Seconds (2007), http://eprint.iacr.org/2007/120.pdf
18. Vaudenay, S., Vuagnoux, M.: Passive-only Key Recovery Attacks on RC4. In: Adams, C., Miri, A., Wiener, M. (eds.) LNCS, vol. 4876, pp. 344–359. Springer, Heidelberg (2007), http://infoscience.epfl.ch/record/115086/files/VV07.pdf

An Improved Security Bound for HCTR

Debrup Chakraborty and Mridul Nandi

Department of Computer Science
CINVESTAV-IPN
Mexico City, Mexico
debrup@cs.cinvestav.mx, mridul.nandi@gmail.com

Abstract. HCTR was proposed by Wang, Feng and Wu in 2005. It is a mode of operation which provides a tweakable strong pseudorandom permutation. Though HCTR is quite an efficient mode, the authors showed a cubic security bound for HCTR which makes it unsuitable for applications where tweakable strong pseudorandom permutations are required. In this paper we show that HCTR has a better security bound than what the authors showed. We prove that the distinguishing advantage of an adversary in distinguishing HCTR and its inverse from a random permutation and its inverse is bounded above by $4.5\sigma^2/2^n$, where n is the block-length of the block-cipher and σ is the number of n-block queries made by the adversary (including the tweak).

1 Introduction

A block-cipher mode of operation is a specific way to use a block-cipher to encrypt messages longer than the block-length of the block-cipher. In the literature there are different modes of operations which provide different kinds of security services like confidentiality, authentication etc. A tweakable enciphering scheme (TES) is a specific kind of mode of operation. They are based on the notion of tweakable block ciphers introduced in [9]. TES are length preserving encryption schemes which can encrypt variable length messages. The security that these modes provide is that of a strong pseudorandom permutation (SPRP), i.e., a TES is considered secure if it is infeasible for any computationally bounded chosen plaintext chosen ciphertext adversary to distinguish between the TES and a random permutation. A TES takes as input a quantity called a tweak other than the message and the key. The tweak is supposed to be a public quantity which enriches the variability of the cipher-text produced.

The first construction of a wide block SPRP was provided by Naor and Reingold [14], but their construction was not a TES as the concept of tweaks came after their construction. A fully defined TES for arbitrary length messages using a block cipher was first presented in [6]. In [6] it was also stated that a possible application area for such encryption schemes could be low level disc encryption, where the encryption/decryption algorithm resides on the disc controller which has access to the disc sectors but has no access to the high level partitions of the disc like directories files, etc. The disc controller encrypts a message before

K. Nyberg (Ed.): FSE 2008, LNCS 5086, pp. 289–302, 2008.

writing it to a specific sector and decrypts the message after reading it from the sector. Additionally it was suggested in [6] that sector addresses can be used as tweaks. Because of the specific nature of this application, a length preserving enciphering scheme is required and under this scenario, a strong pseudorandom permutation can provide the highest possible security.

In the last few years there have been numerous proposals for TES. These proposals fall in three basic categories: Encrypt-Mask-Encrypt type, Hash-ECB-Hash type and Hash-Counter-Hash type. CMC [6], EME [7] and EME* [4] fall under the Encrypt-Mask-Encrypt group. PEP [3], TET [5] and HEH[15] fall under the Hash-ECB-Hash type and XCB [11], HCTR [17], HCH, HCHfp [2], ABL [12] fall under the Hash-Counter-Hash type.

The Encrypt-Mask-Encrypt type constructions require two layers of encryption with a light weight masking layer in between. The other two paradigms require a single layer of encryption sandwiched between two universal hash layers. Thus, the only significant cost for Encrypt-Mask-Encrypt type constructions are the block-cipher calls, whereas for the other two paradigms both block-cipher calls and finite field multiplications are required. More specifically, the Encrypt-Mask-Encrypt paradigm uses about $2m$ block cipher calls for encrypting a m block message and a the other two paradigms require m block-cipher calls and $2m$ field multiplications. Recently, in [16] many different constructions for the Hash-Encrypt-Hash paradigm were proposed using blockwise universal hash functions. One of these proposals called HEH[BRW] uses m field multiplications unlike other constructions of the same category. A detailed comparison of different TES can be found in [2,5,15].

In a recent study [10], some performance data regarding various tweakable enciphering schemes in reconfigurable hardware was reported. This study and the comparisons presented in [2,5,15] indicate that HCTR is one of the most efficient candidates among all proposed TES. But, the security guarantee that the designers of HCTR claimed is insufficient in many practical scenarios. This makes HCTR an uninteresting candidate.

In this paper we show that HCTR provides better security than that claimed by the authors. In fact HCTR provides the same security as other other proposed TES. We consider this result to be important in light of the current activities of the IEEE working group on storage security which is working towards a standard for a wide block TES [8].

The crux of this paper is a security proof for HCTR. The proof technique that we use is a sequence of games as used in [2,5,15]. The previously reported game based proofs for TES performs the final collision analysis on a non-interactive game which runs on a fixed transcript and thus does not depend on the distribution of the queries provided by the adversary. In our proof we do not require the non-interactive game, as we can show that the final collision probabilities are independent of the distribution of the adversarial queries. This observation makes our proof different from the proof in [17] and helps to obtain a better bound.

2 The Construction

In the discussion which follows we shall denote the concatenation of two strings X and Y by $X||Y$. By $|X|$ we shall mean the length of X in bits. $\mathsf{bin}_n(\ell)$ will denote the n bit binary representation of ℓ. For $X, Y \in GF(2^n)$, $X \oplus Y$ and XY will denote addition and multiplication in the field respectively.

HCTR uses two basic building blocks. A universal polynomial hash function and a counter mode of operation. The hash used in case of HCTR is defined as:

$$H_h(X) = X_1 h^{m+1} \oplus X_2 h^m \oplus \ldots \oplus \mathsf{pad}_r(X_m) h^2 \oplus \mathsf{bin}_n(|X|) h \qquad (1)$$

Where h is an n-bit hash key and $X = X_1||X_2||\ldots||X_m$, such that $|X_i| = n$ bits $(i = 1, 2, \ldots m-1)$, $0 < |X_m| \leq n$. The pad function is defined as $\mathsf{pad}_r(X_m) := X_m||0^r$ where $r = n - |X_m|$. Thus, $|\mathsf{pad}_r(X_m)| = n$. If $X = \lambda$, the empty string, we define $H_h(\lambda) = h$. In addition to the hash function, HCTR requires a counter mode of operation. Given an n-bit string S, a sequence $S_1, \ldots, S_m$ is defined, where each S_i depends on S. Given such a sequence and a key K the counter mode is defined as follows.

$$\mathsf{Ctr}_{K,S}(A_1, \ldots, A_m) = (A_1 \oplus E_K(S_1), \ldots, A_m \oplus E_K(S_m)). \qquad (2)$$

Where $S_i = S \oplus \mathsf{bin}_n(i)$. In case the last block A_m is incomplete then $A_m \oplus E_K(S_m)$ in Eq. 2 is replaced by $A_m \oplus \mathsf{drop}_r(E_K(S_m))$, where $r = n - |A_m|$ and $\mathsf{drop}_r(E_K(S_m))$ is the first $(n-r)$ bits of $E_K(S_m)$. The encryption and decryption operations using HCTR are described in Fig. 1, and a high-level description is provided in Fig. 2. If $m = 1$ (when we have one block message), we ignore line 4 in both encryption and decryption algorithm.

HCTR requires m block-cipher calls and $2m+2t+2$ finite field multiplications to encrypt a m block message with a t block tweak. It can be used on any fixed length tweaks. The authors of HCTR prove that the maximum advantage of a

```
Algorithm $\mathbf{E}^T_{K,h}(P_1, \ldots, P_m)$

1. $MM \leftarrow P_1 \oplus H_h(P_2||\ldots||P_m||T)$;
2. $CC \leftarrow E_K(MM)$;
3. $S \leftarrow MM \oplus CC$;

4. $(C_2, \ldots, C_{m-1}, C_m)$
          $\leftarrow \mathsf{Ctr}_{K,S}(P_2, \ldots, P_m)$;

5. $C_1 \leftarrow CC \oplus H_h(C_2||C_3||\ldots||C_m||T)$;
6. return $(C_1, \ldots, C_m)$;
```

```
Algorithm $\mathbf{D}^T_{K,h}(C_1, \ldots, C_m)$

1. $CC \leftarrow C_1 \oplus H_h(C_2||C_3||\ldots||C_m||T)$;
2. $MM \leftarrow E_K^{-1}(CC)$;
3. $S \leftarrow MM \oplus CC$;

4. $(P_2, \ldots, P_{m-1}, P_m)$
          $\leftarrow \mathsf{Ctr}_{K,S}(C_2, \ldots, C_m)$;

5. $P_1 \leftarrow MM \oplus H_h(P_2||\ldots||P_m||T)$;
6. return $(P_1, \ldots, P_m)$;
```

Fig. 1. Encryption using HCTR. K is the block-cipher key, h the hash key and T the tweak.

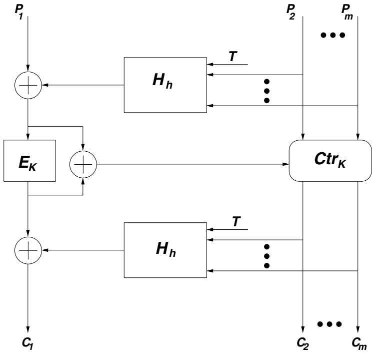


Fig. 2. Encryption using HCTR. Here K is the key for the block cipher $E_K()$ and h is the key for the universal hash function $H_h()$.

chosen plain text and chosen ciphertext adversary in distinguishing HCTR from a random permutation is $\frac{0.5q^2+((2+t)\sigma^2+\sigma^3)}{2^n}$. Where t denotes the length of the tweak and σ denotes the number of blocks of queries made by the adversary. This cubic bound makes HCTR less attractive than other tweakable enciphering schemes all of which are known to have a security bound of the order of $\frac{\sigma^2}{2^n}$.

In a recent work [13] a general construction of tweakable SPRP was reported by using universal hash functions, tweakable block-ciphers and a weak pseudo-random function. The paper [13] also reports a variant of HCTR which comes as an instantiation of their general construction. They claim that this variant of HCTR has a quadratic security bound. But, this variant is quite different and also inefficient from the original specification of HCTR. The variant reported in [13] needs one more block-cipher call than the original HCTR.

3 Improved Bound for HCTR

3.1 Definitions and Notation

The discussion in this section is based on [6]. An n-bit block cipher is a function $E : \mathcal{K} \times \{0,1\}^n \rightarrow \{0,1\}^n$, where $\mathcal{K} \neq \emptyset$ is the key space and for any $K \in \mathcal{K}$, $E(K,.)$ is a permutation. We write $E_K(\,)$ instead of $E(K,.)$.

An adversary A is a probabilistic algorithm which has access to some oracles and which outputs either 0 or 1. Oracles are written as superscripts. The notation $A^{\mathcal{O}_1,\mathcal{O}_2} \Rightarrow 1$ denotes the event that the adversary A, interacts with the oracles $\mathcal{O}_1, \mathcal{O}_2$, and finally outputs the bit 1. In what follows, by the notation $X \xleftarrow{\$} \mathcal{S}$, we will denote the event of choosing X uniformly at random from the finite set $\mathcal{S}$.

Let $\mathrm{Perm}(n)$ denote the set of all permutations on $\{0,1\}^n$. We require $E(,)$ to be a strong pseudorandom permutation. The advantage of an adversary A in breaking the strong pseudorandomness of $E(,)$ is defined in the following manner.

$$\mathbf{Adv}_E^{\pm\mathrm{prp}}(A) = \Big|\Pr\Big[K \xleftarrow{\$} \mathcal{K} : A^{E_K(\,),E_K^{-1}(\,)} \Rightarrow 1\Big] - \Pr\Big[\pi \xleftarrow{\$} \mathrm{Perm}(n) : A^{\pi(\,),\pi^{-1}(\,)} \Rightarrow 1\Big]\Big|.$$

A tweakable enciphering scheme is a function $\mathbf{E} : \mathcal{K} \times \mathcal{T} \times \mathcal{M} \rightarrow \mathcal{M}$, where $\mathcal{K} \neq \emptyset$ and $\mathcal{T} \neq \emptyset$ are the key space and the tweak space respectively. The message and the cipher spaces are $\mathcal{M}$. For HCTR we have $\mathcal{M} = \cup_{i>n}\{0,1\}^i$. We shall write $\mathbf{E}_K^T(.)$ instead of $\mathbf{E}(K, T, .)$. The inverse of an enciphering scheme is $\mathbf{D} = \mathbf{E}^{-1}$ where $X = \mathbf{D}_K^T(Y)$ if and only if $\mathbf{E}_K^T(X) = Y$.

Let $\mathrm{Perm}^{\mathcal{T}}(\mathcal{M})$ denote the set of all functions $\boldsymbol{\pi} : \mathcal{T} \times \mathcal{M} \rightarrow \mathcal{M}$ where $\boldsymbol{\pi}(\mathcal{T}, .)$ is a length preserving permutation. Such a $\boldsymbol{\pi} \in \mathrm{Perm}^{\mathcal{T}}(\mathcal{M})$ is called a tweak indexed permutation. For a tweakable enciphering scheme $\mathbf{E} : \mathcal{K} \times \mathcal{T} \times \mathcal{M} \rightarrow \mathcal{M}$, we define the advantage an adversary A has in distinguishing $\mathbf{E}$ and its inverse from a random tweak indexed permutation and its inverse in the following manner.

$$\mathbf{Adv}_{\mathbf{E}}^{\pm\widetilde{\mathrm{prp}}}(A) = \Big|\Pr\Big[K \xleftarrow{\$} \mathcal{K} : A^{\mathbf{E}_K(.,.),\mathbf{E}_K^{-1}(.,.)} \Rightarrow 1\Big] \\ - \Pr\Big[\boldsymbol{\pi} \xleftarrow{\$} \mathrm{Perm}^{\mathcal{T}}(\mathcal{M}) : A^{\boldsymbol{\pi}(.,.),\boldsymbol{\pi}^{-1}(.,.)} \Rightarrow 1\Big]\Big|. \tag{3}$$

Here, $\boldsymbol{\pi} \xleftarrow{\$} \mathrm{Perm}^{\mathcal{T}}(\mathcal{M})$ means that for each ℓ such that $\{0,1\}^\ell \subseteq \mathcal{M}$ and $T \in \mathcal{T}$ we choose a tweakable random permutation π^T from $\mathrm{Perm}(\ell)$ independently. We define $\mathbf{Adv}_{\mathbf{E}}^{\pm\widetilde{\mathrm{prp}}}(q, \sigma)$ by $\max_A \mathbf{Adv}_{\mathbf{E}}^{\pm\widetilde{\mathrm{prp}}}(A)$ where maximum is taken over all adversaries which makes at most q queries having at most σ many blocks. For a computational advantage we define $\mathbf{Adv}_{\mathbf{E}}^{\pm\widetilde{\mathrm{prp}}}(q, \sigma, t)$ by $\max_A \mathbf{Adv}_{\mathbf{E}}^{\pm\widetilde{\mathrm{prp}}}(A)$. In addition to the previous restrictions on A, he can run in time at most t.

Pointless queries: Let T, P and C represent tweak, plaintext and ciphertext respectively. We assume that an adversary never repeats a query, i.e., it does not ask the encryption oracle with a particular value of (T, P) more than once and neither does it ask the decryption oracle with a particular value of (T, C) more than once. Furthermore, an adversary never queries its deciphering oracle with (T, C) if it got C in response to an encipher query (T, P) for some P. Similarly, the adversary never queries its enciphering oracle with (T, P) if it got P as a response to a decipher query of (T, C) for some C. These queries are called *pointless* as the adversary knows what it would get as responses for such queries.

The notation HCTR$[E]$ denotes a tweakable enciphering scheme, where the n-bit block cipher E is used in the manner specified by HCTR. We will use the notation $\mathbf{E}_\pi$ as a shorthand for HCTR[Perm(n)] and $\mathbf{D}_\pi$ will denote the inverse of $\mathbf{E}_\pi$. Thus, the notation $A^{\mathbf{E}_\pi,\mathbf{D}_\pi}$ will denote an adversary interacting with the oracles $\mathbf{E}_\pi$ and $\mathbf{D}_\pi$.

3.2 Statement of Results

The following theorem specifies the security of HCTR.

Theorem 1. *Fix n, σ to be positive integers and an n-bit block cipher $E : \mathcal{K} \times \{0,1\}^n \rightarrow \{0,1\}^n$. Then*

$$\mathbf{Adv}_{\mathrm{HCTR[Perm}(n)]}^{\pm\widetilde{\mathrm{prp}}}(\sigma) \leq \frac{4.5\sigma^2}{2^n}. \tag{4}$$

$$\mathbf{Adv}^{\pm\widetilde{\mathrm{prp}}}_{\mathrm{HCTR}[E]}(\sigma,t) \leq \frac{4.5\sigma^2}{2^n} + \mathbf{Adv}^{\pm\mathrm{prp}}_{E}(\sigma,t') \tag{5}$$

where $t' = t + O(\sigma)$*.*

The above result and its proof is similar to previous work (see for example [6,7,3]). As mentioned in [6], Equation (5) embodies a standard way to pass from the information theoretic setting to the complexity theoretic setting.

For proving (4), we need to consider an adversary's advantage in distinguishing a tweakable enciphering scheme $\mathbf{E}$ from an oracle which simply returns random bit strings. This advantage is defined in the following manner.

$$\begin{aligned}\mathbf{Adv}^{\pm\mathrm{rnd}}_{\mathrm{HCTR}[\mathrm{Perm}(n)]}(A) = \Big|&\Pr\Big[\pi \xleftarrow{\$} \mathrm{Perm}(n) : A^{\mathbf{E}_\pi,\mathbf{D}_\pi} \Rightarrow 1\Big] \\ &- \Pr\Big[A^{\$(.,.),\$(.,.)} \Rightarrow 1\Big]\Big| \end{aligned} \tag{6}$$

where $\$(.,M)$ or $\$(.,C)$ returns independently distributed random bits of length $|M|$ or $|C|$ respectively. The basic idea of proving (4) is as follows.

$$\begin{aligned}\mathbf{Adv}^{\pm\widetilde{\mathrm{prp}}}_{\mathrm{HCH}[\mathrm{Perm}(n)]}(A) &= \Big(\Pr\Big[\pi \xleftarrow{\$} \mathrm{Perm}(n) : A^{\mathbf{E}_\pi,\mathbf{D}_\pi} \Rightarrow 1\Big] \\ &\quad - \Pr\Big[\boldsymbol{\pi} \xleftarrow{\$} \mathrm{Perm}^{\mathcal{T}}(\mathcal{M}) : A^{\boldsymbol{\pi}(.,.),\boldsymbol{\pi}^{-1}(.,.)} \Rightarrow 1\Big]\Big) \\ &= \Big(\Pr\Big[\pi \xleftarrow{\$} \mathrm{Perm}(n) : A^{\mathbf{E}_\pi,\mathbf{D}_\pi} \Rightarrow 1\Big] \\ &\quad - \Pr\Big[A^{\$(.,.),\$(.,.)} \Rightarrow 1\Big]\Big) \\ &\quad + \Big(\Pr\Big[A^{\$(.,.),\$(.,.)} \Rightarrow 1\Big] \\ &\quad - \Pr\Big[\boldsymbol{\pi} \xleftarrow{\$} \mathrm{Perm}^{\mathcal{T}}(\mathcal{M}) : A^{\boldsymbol{\pi}(.,.),\boldsymbol{\pi}^{-1}(.,.)} \Rightarrow 1\Big]\Big) \\ &\leq \mathbf{Adv}^{\pm\mathrm{rnd}}_{\mathrm{HCH}[\mathrm{Perm}(n)]}(A) + \binom{q}{2}\frac{1}{2^n} \end{aligned} \tag{7}$$

where q is the number of queries made by the adversary. For a proof of the last inequality see [6]. Thus, the main task of the proof now reduces to obtaining an upper bound on $\mathbf{Adv}^{\pm\mathrm{rnd}}_{\mathrm{HCTR}[\mathrm{Perm}(n)]}(\sigma)$. In section 4 we prove that

$$\mathbf{Adv}^{\pm\mathrm{rnd}}_{\mathrm{HCTR}[\mathrm{Perm}(n)]}(\sigma) \leq \frac{4\sigma^2}{2^n}. \tag{8}$$

Using equation (8) and (7) we obtain equation (4).

4 The Game Sequence

We shall model the interaction of the adversary with HCTR by a sequence of games. We shall start with the game HCTR1 which describes the mode HCTR, and with small changes we shall reach the game RAND2 which will represent an oracle which returns just random strings and we shall bound the advantage of an adversary in distinguishing between the games HCTR1 and RAND1. Where G represents a game by $\Pr[A^G \Rightarrow 1]$ we shall mean the probability that A outputs 1 by interacting with the game G. Next we describe the games.

Game HCTR1: In HCTR1, the adversary interacts with $\mathbf{E}_\pi$ and $\mathbf{D}_\pi$ where π is a randomly chosen permutation from Perm(n). Instead of initially choosing π, we build up π in the following manner.

Initially π is assumed to be undefined everywhere. When $\pi(X)$ is needed, but the value of π is not yet defined at X, then a random value is chosen among the available range values. Similarly when $\pi^{-1}(Y)$ is required and there is no X yet defined for which $\pi(X) = Y$, we choose a random value for $\pi^{-1}(Y)$ from the available domain values.

The domain and range of π are maintained in two sets *Domain* and *Range*, and $\overline{Domain}$ and $\overline{Range}$ are the complements of *Domain* and *Range* relative to $\{0,1\}^n$. The game HCTR1 is shown in Figure 3. The figure shows the subroutines Ch-π, Ch-π^{-1}, the initialization steps and how the game responds to a encipher/decipher query of the adversary. The i^{th} query of the adversary depends on its previous queries, the responses to those queries and on some coins of the adversary. When $l^s = n$, we ignore the line 103 to line 109.

The game HCTR1 accurately represents the attack scenario, and by our choice of notation, we can write

$$\Pr[A^{\mathbf{E}_\pi, \mathbf{D}_\pi} \Rightarrow 1] = \Pr[A^{\text{HCTR}_1} \Rightarrow 1]. \tag{9}$$

Game RAND1: We modify HCTR1 by deleting the boxed entries in HCTR1 and call the modified game as RAND1. By deleting the boxed entries it cannot be guaranteed that π is a permutation as though we do the consistency checks but we do not reset the values of Y (in Ch-π) and X (in Ch-π^{-1}). Thus, the games HCTR1 and RAND1 are identical apart from what happens when the bad flag is set. By using the result from [1], we obtain

$$|\Pr[A^{\text{HCTR1}} \Rightarrow 1] - \Pr[A^{\text{RAND1}} \Rightarrow 1]| \leq \Pr[A^{\text{RAND1}} \text{ sets } \mathsf{bad}] \tag{10}$$

Another important thing to note is that in RAND1 in line 103, for a encryption query CC^s (and MM^s for a decryption query) gets set to a random n bit string. Similarly 105 and 108 Z_i^s gets set to random values. Thus the the adversary gets random strings in response to both his encryption and decryption queries. Hence,

$$\Pr[A^{\text{RAND1}} \Rightarrow 1] = \Pr[A^{\$(.,.),\$(.,.)} \Rightarrow 1] \tag{11}$$

So using Equations (6), (10) and (11) we get

$$\begin{aligned}
\mathbf{Adv}^{\pm\text{rnd}}_{\text{HCTR}[\text{Perm}(n)]}(A) &= |\Pr[A^{\mathbf{E}_\pi, \mathbf{D}_\pi} \Rightarrow 1] - \Pr[A^{\$(.,.),\$(.,.)} \Rightarrow 1]| && (12)\\
&= |\Pr[A^{\text{HCTR1}} \Rightarrow 1] - \Pr[A^{\text{RAND1}} \Rightarrow 1]| \\
&\leq \Pr[A^{\text{RAND1}} \text{ sets } \mathsf{bad}] && (13)
\end{aligned}$$

```
Subroutine Ch-$\pi(X)$
    $Y \xleftarrow{\$} \{0,1\}^n$; if $Y \in Range$ then bad $\leftarrow$ true; $\boxed{Y \xleftarrow{\$} \overline{Range}}$; endif;
    if $X \in Domain$ then bad $\leftarrow$ true; $\boxed{Y \leftarrow \pi(X)}$; endif
    $\pi(X) \leftarrow Y$; $Domain \leftarrow Domain \cup \{X\}$; $Range \leftarrow Range \cup \{Y\}$; return$(Y)$;

Subroutine Ch-$\pi^{-1}(Y)$
    $X \xleftarrow{\$} \{0,1\}^n$; if $X \in Domain$, bad $\leftarrow$ true; $\boxed{X \xleftarrow{\$} \overline{Domain}}$; endif;
    if $Y \in Range$ then bad $\leftarrow$ true; $\boxed{X \leftarrow \pi^{-1}(Y)}$; endif;
    $\pi(X) \leftarrow Y$; $Domain \leftarrow Domain \cup \{X\}$; $Range \leftarrow Range \cup \{Y\}$; return$(X)$;

Initialization:
    for all $X \in \{0,1\}^n$ $\pi(X) =$ undef endfor
    bad = false
```

Respond to the s^{th} query as follows: (Assume $l^s = n(m^s - 1) + r^s$, with $0 \le r^s < n$.)

```
Encipher query: Enc$(T^s; P_1^s, P_2^s, \ldots P_{m^s}^s)$

$MM^s \leftarrow P_1^s \oplus H_h(P_2^s || \ldots || P_m^s || T^s)$;
$CC^s \leftarrow$ Ch-$\pi(MM^s)$;

$S^s \leftarrow MM^s \oplus CC^s$;
for $i = 1$ to $m^s - 2$,
    $Z_i^s \leftarrow$ Ch-$\pi(S^s \oplus \mathsf{bin}_n(i))$;
    $C_{i+1}^s \leftarrow P_{i+1}^s \oplus Z_i^s$;
end for
$Z_{m^s}^s \leftarrow$ Ch-$\pi(S^s \oplus \mathsf{bin}_n(m^s - 1))$;
$C_{m^s}^s \leftarrow P_{m^s}^s \oplus \mathsf{drop}_{n-r^s}(Z_{m^s}^s)$;

$C_1^s \leftarrow CC^s \oplus H_h(C_2^s || \ldots || C_{m^s}^s || T^s)$;
return $C_1^s || C_2^s || \ldots || C_{m^s}^s$
```

```
Decipher query: Dec$(C_1^s, C_2^s, \ldots, C_{m^s}^s, T^s)$

$CC^s \leftarrow C_1^s \oplus H_h(C_2^s || \ldots || C_m^s || T^s)$;
$MM^s \leftarrow$ Ch-$\pi^{-1}(CC^s)$

$S^s \leftarrow MM^s \oplus CC^s$;
for $i = 1$ to $m^s - 2$,
    $Z_i^s \leftarrow$ Ch-$\pi(S^s \oplus \mathsf{bin}_n(i))$;
    $P_{i+1}^s \leftarrow C_{i+1}^s \oplus Z_i^s$;
end for
$Z_{m^s}^s \leftarrow$ Ch-$\pi(S^s \oplus \mathsf{bin}_n(m^s - 1))$;
$P_{m^s}^s \leftarrow C_{m^s}^s \oplus \mathsf{drop}_{n-r^s}(Z_{m^s}^s)$;
$P_1^s \leftarrow MM^s \oplus H_h(P_2^s || \ldots || P_m^s || T^s)$;

return $P_2^s || \ldots || P_{m^s}^s$
```

Fig. 3. Games HCTR1 and RAND1

Game RAND2: Now we make some subtle changes in the game RAND1 to get a new game RAND2 which is described in Figure 4. In game RAND1 the permutation was not maintained and a call to the permutation was responded by returning random strings, so in Game RAND2 we no more use the subroutines Ch-π and Ch-π^{-1}. Here we immediately return random strings to the adversary in response to his encryption or decryption queries. Later in the finalization step we adjust variables and maintain multi sets $\mathcal{D}$ and $\mathcal{R}$ where we list the elements that were supposed to be inputs and outputs of the permutation. In the second phase of the finalization step, we check for collisions in the sets $\mathcal{D}$ and $\mathcal{R}$, and in the event of a collision we set the bad flag to true.

Game RAND1 and Game RAND2 are indistinguishable to the adversary, as in both cases he gets random strings in response to his queries. Also, the probability

```
Respond to the $s^{th}$ adversary query as follows:

ENCIPHER QUERY $\mathsf{Enc}(T^s; P_1^s, P_2^s, \ldots, P_{m^s}^s)$
    $ty^s = \mathsf{Enc}$; $C_1^s||C_2^s||\ldots||C_{m^s-1}^s||D_{m^s}^s \xleftarrow{\$} \{0,1\}^{nm^s}$;
    $C_{m^s}^s \leftarrow \mathsf{drop}_{n-r^s}(D_{m^s})$ return $C_1^s||C_2^s||\ldots||C_{m^s}^s$;

DECIPHER QUERY $\mathsf{Dec}(T^s; C_1^s, C_2^s, \ldots, C_{m^s}^s)$
    $ty^s = \mathsf{Dec}$; $P_1^s||P_2^s||\ldots||P_{m_s-1}^s||V_{m^s}^s \xleftarrow{\$} \{0,1\}^{nm^s}$;
    $P_{m^s}^s \leftarrow \mathsf{drop}_{n-r^s}(V_{m^s})$ return $P_1^s||P_2^s||\ldots||P_{m^s}^s$;

Finalization:
Case $ty^s = \mathsf{Enc}$:

  $MM^s \leftarrow P_1^s \oplus H_h(P_2^s||\ldots||P_m^s||T^s)$;
  $CC^s \leftarrow C_1^s \oplus H_h(C_2^s||\ldots||C_m^s||T^s)$;
  $S^s \leftarrow MM^s \oplus CC^s$;
  $\mathcal{D} \leftarrow \mathcal{D} \cup \{MM^s\}$;
  $\mathcal{R} \leftarrow \mathcal{R} \cup \{CC^s\}$;
  for $i = 2$ to $m^s - 1$,
    $Y_i^s \leftarrow C_i^s \oplus P_i^s$;
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{S^s \oplus \mathsf{bin}_n(i-1)\}$;
    $\mathcal{R} \leftarrow \mathcal{R} \cup \{Y_i^s\}$;
  end for
  $Y_{m^s}^s \leftarrow D_{m^s}^s \oplus P_{m^s}^s$
  $\mathcal{D} \leftarrow \mathcal{D} \cup \{S^s \oplus \mathsf{bin}_n(m^s-1)\}$;
  $\mathcal{R} \leftarrow \mathcal{R} \cup \{Y_{m^s}^s\}$;

Case $ty^s = \mathsf{Dec}$:

  $MM^s \leftarrow P_1^s \oplus H_h(P_2^s||\ldots||P_m^s||T^s)$;
  $CC^s \leftarrow C_1^s \oplus H_h(C_2^s||\ldots||C_m^s||T^s)$;
  $S^s \leftarrow MM^s \oplus CC^s$;
  $\mathcal{D} \leftarrow \mathcal{D} \cup \{MM^s\}$;
  $\mathcal{R} \leftarrow \mathcal{R} \cup \{CC^s\}$;
  for $i = 2$ to $m^s - 1$,
    $Y_i^s \leftarrow C_i^s \oplus P_i^s$;
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{S^s \oplus \mathsf{bin}_n(i-1)\}$;
    $\mathcal{R} \leftarrow \mathcal{R} \cup \{Y_i^s\}$;
  end for
  $Y_{m^s}^s \leftarrow V_{m^s}^s \oplus C_{m^s}^s$
  $\mathcal{D} \leftarrow \mathcal{D} \cup \{S^s \oplus \mathsf{bin}_n(m^s-1)\}$;
  $\mathcal{R} \leftarrow \mathcal{R} \cup \{Y_{m^s}^s\}$;

SECOND PHASE
    bad = false;
    if (some value occurs more than once in $\mathcal{D}$) then bad = true endif;
    if (some value occurs more than once in $\mathcal{R}$) then bad = true endif.
```

Fig. 4. Game RAND2

with which RAND1 sets bad is same as the probability with which RAND2 sets bad. Thus we get:

$$\Pr[A^{\mathrm{RAND1}} \text{ sets } \mathsf{bad}] = \Pr[A^{\mathrm{RAND2}} \text{ sets } \mathsf{bad}] \qquad (14)$$

Thus from Equations (13) and (14) we obtain

$$\mathbf{Adv}^{\pm\mathrm{rnd}}_{\mathrm{HCTR}[\mathrm{Perm}(n)]}(A) \leq \Pr[A^{\mathrm{RAND2}} \text{ sets } \mathsf{bad}] \qquad (15)$$

Now our goal would be to bound $\Pr[A^{\mathrm{RAND2}}$ sets $\mathsf{bad}]$. We notice that in Game RAND2 the bad flag is set when there is a collision in either of the sets $\mathcal{D}$ or $\mathcal{R}$. So if COLLD and COLLR denote the events of a collision in $\mathcal{D}$ and $\mathcal{R}$ respectively then we have

$$\Pr[A^{\mathrm{RAND2}} \text{ sets } \mathsf{bad}] \leq \Pr[\mathsf{COLLR}] + \Pr[\mathsf{COLLD}]$$

In many previously reported game based proofs for strong pseudorandom permutations including the proof given in [17], the final collision analysis is done on a non-interactive game. The non-interactive game is generally obtained by eliminating the randomness present in the distribution of the queries presented by the adversary. To achieve this the final non-interactive game runs on a fixed transcript which maximizes the probability of bad being set to true. In this case as we will soon see, such a de-randomization is not required. Because of the specific structure of the game RAND2 the probability COLLR and COLLD would be independent of the distribution of the queries supplied by the adversary, hence a final collision analysis can be done on the game RAND2 itself.

4.1 Bounding Collision Probability in $\mathcal{D}$ and $\mathcal{R}$

In the analysis we consider the sets $\mathcal{D}$ and $\mathcal{R}$ to consist of the formal variables instead of their values. For example, whenever we set $\mathcal{D} \leftarrow \mathcal{D} \cup \{X\}$ for some variable X we think of it as setting $\mathcal{D} \leftarrow \mathcal{D} \cup \{\text{“}X\text{”}\}$ where “X” is the name of that formal variable. This is the same technique as used in [6]. Our goal is to bound the probability that two formal variables in the sets $\mathcal{D}$ and $\mathcal{R}$ take the same value. After q queries of the adversary where the s^{th} query has m^s blocks of plaintext or ciphertext and t block of tweak, then the sets $\mathcal{D}$ and $\mathcal{R}$ can be written as follows:

$$\begin{aligned}
\text{Elements in } \mathcal{D}: \quad & MM^s = P_1^s \oplus Q^s, \\
& S_j^s = S^s \oplus \mathsf{bin}_n(j) = (P_1^s \oplus C_1^s) \oplus (Q^s \oplus B^s \oplus \mathsf{bin}_n(j)), \\
& \text{where } Q^s = H_h(P_2^s \parallel \cdots \parallel P_{m^s}^s \parallel T^s) \text{ and} \\
& B^s = H_h(C_2^s \parallel \cdots \parallel C_{m^s}^s \parallel T^s), \\
& 1 \le s \le q,\ 1 \le i \le m^s - 1,
\end{aligned}$$

$$\begin{aligned}
\text{Elements in } \mathcal{R}: \quad & CC^s = C_1^s \oplus B^s, \\
& Y_i^s = C_i^s \oplus P_i^s, \\
& 2 \le i \le m^s,\ 1 \le s \le q.
\end{aligned}$$

Before we present the collision analysis let us identify the random variables based on which the probability of collision would be computed. In game RAND2 the hash key h is selected uniformly from the set $\{0,1\}^n$. The outputs that the adversary receives are also uniformly distributed, and are independent of the previous queries supplied by the adversary and the outputs obtained by the adversary. The i^{th} query supplied by the adversary may depend on the previous outputs obtained by the adversary, but as the output of GAME2 is not dependent in any way on the hash key h thus the queries supplied by the adversary are independent of h.

We consider T^s as t n-bit blocks. Thus, for any s, $H_h(P_2^s||\cdots||P_{m^s}^s||T^s)$ or $H_h(C_2^s||\cdots||C_{m^s}^s||T^s)$ has degree at most $m^s + t$. We denote $\sigma = qt + \sum_s m^s$.

We denote $\ell^{s,s'} = \max\{m^s, m^{s'}\} + t$. Since $\ell^{s,s'} \leq m^s + m^{s'} + t$, we have the following inequality

$$\begin{aligned}\sum_{1\leq s<s'\leq q} \ell^{s,s'} &\leq \binom{q}{2}t + \sum_{1\leq s<s'\leq q} (m^s + m^{s'}) \\ &\leq \binom{q}{2}t + (q-1)(\sigma - qt) \\ &\leq (q-1)\sigma + \frac{qt(q-1)}{2} - qt(q-1) \\ &\leq (q-1)\sigma.\end{aligned}$$

We also note that the response of encryption or decryption query are completely independent of h (the poly hash key). Thus, inputs of $H_h(\cdot)$ for each query are independent with h. So we can use the fundamental theorem of algebra to claim that the probability that h is a root of a d degree polynomial is at most $d/2^n$ where h is chosen uniformly and independently from the coefficient of the polynomial (which is true in case of H_h in RAND2 game).

First we consider the collisions in $\mathcal{R}$.

- We first consider collision among CC^s. Let $s' \neq s$. Now, $\Pr[CC^s = CC^{s'}] \leq \ell^{s,s'}/2^n$ where the probability is computed under the uniform choice of $h \in \{0,1\}^n$. We know that $CC^s \oplus CC^{s'}$ is a non-zero polynomial of h with degree at most $\ell^{s,s'}$. By using fundamental theorem of algebra we have the above bound for the collision probability. Thus,

$$\begin{aligned}\Pr[CC^s = CC^{s'} : \text{for some } 1 \leq s < s' \leq q] &\leq \sum_{1\leq s<s'\leq q} \frac{\ell^{s,s'}}{2^n} \\ &\leq \frac{(q-1)\sigma}{2^n}. \qquad (16)\end{aligned}$$

  Similarly we can compute collision probability between Y_i^s and $CC^{s'}$. For each s', there are $(\sigma - qt - q)$ many Y_i^s's. For each such choice, $\Pr[CC^{s'} = Y_i^s] \leq (m^{s'} + t)/2^n$. Thus,

$$\begin{aligned}&\Pr[CC^{s'} = Y_i^s : \text{ for some } 1 \leq s \neq s' \leq q, 2 \leq i \leq m^s] \\ &\qquad\leq \sum_{1\leq s'\leq q} \frac{(\sigma - qt - q)(m^{s'} + t)}{2^n} \\ &\qquad\leq \sigma^2/2^n. \qquad (17)\end{aligned}$$

- Now we consider collision among Y_i^s, $2 \leq i \leq m^s$, $1 \leq s \leq q$. For the pairs $(Y_i^s, Y_{i'}^{s'})$ with $s' \leq s$ and $(s,i) \neq (s',i')$, the collision probability is $1/2^n$, since either P^s or C^s is chosen uniformly and independently from the rest of the variables. There are $\binom{\sigma-qt-q}{2}$ pairs of this form. Thus,

$$\begin{aligned}&\Pr[Y_i^s = Y_{i'}^{s'} : \text{ for some } 1 \leq s \leq s' \leq q, 1 \leq i,i' \leq q, (s,i) \neq (s',i')] \\ &\qquad\leq \binom{\sigma - qt - q}{2}/2^n. \qquad (18)\end{aligned}$$

Combining equation (16), (17) and (18) we obtain

$$\Pr[\text{COLLR}] \leq \frac{4\sigma^2}{2^{n+1}}. \tag{19}$$

Now we consider collision in domain $\mathcal{D}$.

- Similar to equations (16) and (17), we have

$$\begin{aligned}\Pr[MM^s = MM^{s'} : \text{ for some } 1 \leq s < s' \leq q] &\leq \sum_{1 \leq s < s' \leq q} \frac{\ell^{s,s'}}{2^n} \\ &\leq (q-1)\sigma/2^n. \end{aligned} \tag{20}$$

$$\Pr[MM^{s'} = S_i^s : \text{ for some } 1 \leq s \neq s' \leq q, 2 \leq i \leq m^s] \leq \sigma^2/2^n. \tag{21}$$

- Now we consider collision among $S_i^s = S^s \oplus \mathsf{bin}_n(i)$, $2 \leq i \leq m^s$, $1 \leq s \leq q$. Note that, $S_i^s = S_{i'}^{s'}$ implies that $(P_1^s \oplus C_1^s) \oplus (Q^s \oplus B^s \oplus \mathsf{bin}_n(i)) = (P_1^{s'} \oplus C_1^{s'}) \oplus (Q^{s'} \oplus B^{s'} \oplus \mathsf{bin}_n(i'))$. Let $s' \leq s$ and $(s,i) \neq (s',i')$. Thus, either C_1^s (in case s^{th} query is encryption) or P_1^s (in case s^{th} query is decryption) is uniformly and independently distributed with all other variables stated in the above equality. Thus, the collision probability is $1/2^n$. Since there are $\binom{\sigma-qt-q}{2}$ pairs of this form, we have

$$\begin{aligned}\Pr[S_i^s = S_{i'}^{s'} : \text{ for some } 1 \leq s \leq s' \leq q, 1 \leq i,i' \leq q, (s,i) \neq (s',i')] \\ \leq \binom{\sigma - qt - q}{2}/2^n. \end{aligned} \tag{22}$$

The equations (20), (21) and (22) imply the following similar bound for domain collision probability.

$$\Pr[\text{COLLD}] \leq \frac{4\sigma^2}{2^{n+1}}. \tag{23}$$

Combining the domain and range collision probabilities, we obtain the probability of bad being set to true in RAND2 to be at most $8\sigma^2/2^{n+1}$. Thus, by using equations (19) and (23), we have

$$\mathbf{Adv}^{\pm\text{rnd}}_{\text{HCTR}[\text{Perm}(n)]}(A) \leq \frac{4\sigma^2}{2^n}. \tag{24}$$

5 Discussions

Why our bound is different from [17]: The analysis that we perform is very similar to that presented in [17]. As stated earlier, the authors in [17] presents their collision analysis on a non-interactive game where the plain texts and ciphertexts are fixed. Thus they obtain a different bound for the probability of collisions between S_i^s and $S_{i'}^{s'}$. As they consider the plaintext and ciphertexts to be fixed thus they conclude that the probability of collision between each pair is less than $\ell/2^n$, where ℓ is the maximum length of a query supplied by the adversary. Thus according to their analysis they obtain

$$\Pr[S_i^s = S_{i'}^{s'} : \text{ for some } 1 \leq s \leq s' \leq q, 1 \leq i, i' \leq q, (s,i) \neq (s',i')] \\ \leq \ell \binom{\sigma - qt - q}{2} / 2^n. \quad (25)$$

This term contributes to the cubic security bound reported in [17].

The bound claimed in [13]: In [13] a improved bound provided of a variant of HCTR. Firstly, the variant uses one more block-cipher call than HCTR making it less efficient than the original construction. Secondly, they claim that the security bound of modified HCTR is $O(\frac{q^2\ell^2}{2^n})$, where ℓ is the maximum query length. This bound is uniformly larger than our bound.

6 Conclusion

We provided a improved security analysis of the HCTR mode of operation. This work thus establish that HCTR provides same security guarantee as provided by CMC, EME, EME*, XCB, PEP, HCH, TET, and HEH (to our knowledge these are the only TES with a security proof).

References

1. Bellare, M., Rogaway, P.: Code-based game-playing proofs and the security of triple encryption. Cryptology ePrint Archive, Report 2004/331 (2004), http://eprint.iacr.org/
2. Chakraborty, D., Sarkar, P.: HCH: A new tweakable enciphering scheme using the hash-encrypt-hash approach. In: Barua, R., Lange, T. (eds.) INDOCRYPT 2006. LNCS, vol. 4329, pp. 287–302. Springer, Heidelberg (2006), http://eprint.iacr.org/2007/028
3. Chakraborty, D., Sarkar, P.: A new mode of encryption providing a tweakable strong pseudo-random permutation. In: Robshaw, M.J.B. (ed.) FSE 2006. LNCS, vol. 4047, pp. 293–309. Springer, Heidelberg (2006)
4. Halevi, S.: EME*: Extending EME to Handle Arbitrary-Length Messages with Associated Data. In: Canteaut, A., Viswanathan, K. (eds.) INDOCRYPT 2004. LNCS, vol. 3348, pp. 315–327. Springer, Heidelberg (2004)
5. Halevi, S.: Invertible universal hashing and the tet encryption mode. In: Menezes, A. (ed.) CRYPTO 2007. LNCS, vol. 4622, pp. 412–429. Springer, Heidelberg (2007)
6. Halevi, S., Rogaway, P.: A tweakable enciphering mode. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 482–499. Springer, Heidelberg (2003)
7. Halevi, S., Rogaway, P.: A parallelizable enciphering mode. In: Okamoto, T. (ed.) CT-RSA 2004. LNCS, vol. 2964, pp. 292–304. Springer, Heidelberg (2004)
8. IEEE Security in Storage Working Group (SISWG). PRP modes comparison IEEE, March 2007, pp. 1619–2. IEEE Computer Society, Los Alamitos (2007), http://siswg.org/

9. Liskov, M., Rivest, R.L., Wagner, D.: Tweakable block ciphers. In: Yung, M. (ed.) CRYPTO 2002. LNCS, vol. 2442, pp. 31–46. Springer, Heidelberg (2002)
10. Mancillas-López, C., Chakraborty, D., Rodríguez-Henríquez, F.: Efficient implementations of some tweakable enciphering schemes in reconfigurable hardware. In: Srinathan, K., Rangan, C.P., Yung, M. (eds.) INDOCRYPT 2007. LNCS, vol. 4859, pp. 414–424. Springer, Heidelberg (2007)
11. McGrew, D.A., Fluhrer, S.R.: The extended codebook (XCB) mode of operation. Cryptology ePrint Archive, Report 2004/278 (2004), http://eprint.iacr.org/
12. McGrew, D.A., Viega, J.: Arbitrary block length mode (2004), http://grouper.ieee.org/groups/1619/email/pdf00005.pdf
13. Minematsu, K., Matsushima, T.: Tweakable enciphering schemes from hash-sum-expansion. In: Srinathan, K., Rangan, C.P., Yung, M. (eds.) INDOCRYPT 2007. LNCS, vol. 4859, pp. 252–267. Springer, Heidelberg (2007)
14. Naor, M., Reingold, O.: A pseudo-random encryption mode, http://www.wisdom.weizmann.ac.il/~naor
15. Sarkar, P.: Improving upon the TET mode of operation. In: Srinathan, K., Rangan, C.P., Yung, M. (eds.) INDOCRYPT 2007. LNCS, vol. 4859, pp. 180–192. Springer, Heidelberg (2007)
16. Sarkar, P.: Efficient tweakable enciphering schemes from (block-wise) universal hash functions. Cryptology ePrint Archive, Report 2008/004 (2008), http://eprint.iacr.org/
17. Wang, P., Feng, D., Wu, W.: HCTR: A variable-input-length enciphering mode. In: Feng, D., Lin, D., Yung, M. (eds.) CISC 2005. LNCS, vol. 3822, pp. 175–188. Springer, Heidelberg (2005)

How to Encrypt with a Malicious Random Number Generator

Seny Kamara[1],* and Jonathan Katz[2],**

[1] Johns Hopkins University
seny@cs.jhu.edu
[2] University of Maryland
jkatz@cs.umd.edu

Abstract. Chosen-plaintext attacks on private-key encryption schemes are currently modeled by giving an adversary access to an oracle that encrypts a given message m using random coins that are generated *uniformly at random* and *independently* of anything else. This leaves open the possibility of attacks in case the random coins are poorly generated (e.g., using a faulty random number generator), or are under partial adversarial control (e.g., when encryption is done by lightweight devices that may be captured and tampered with).

We introduce new notions of security modeling such attacks, propose two concrete schemes meeting our definitions, and show generic transformations for achieving security in this context.

Keywords: Private-key encryption, random number generation.

1 Introduction

Security against chosen-plaintext attacks (CPA-security) [10,2,12] is, nowadays, considered a minimal notion of security that any private-key encryption scheme deployed in practice should satisfy. (We defer for now any discussion of security against chosen-ciphertext attacks, though we will consider such attacks later.) Very roughly speaking, CPA-security means that given a challenge ciphertext generated using an unknown key K, a computationally-bounded adversary cannot recover any partial information about the underlying plaintext even if it is given access to an encryption oracle that returns an encryption (using the same key K) of any message m provided by the adversary. This "encryption oracle" is meant, in part, to model potential real-world actions of an adversary that might influence the honest sender (holding the key K) to encrypt certain messages that are (partially or entirely) under the adversary's control.

It is not hard to see that any scheme secure with respect to chosen-plaintext attacks must be *probabilistic*. Furthermore, it is by now well-understood how to construct CPA-secure schemes under the assumption that the sender is able

* Supported in part by the Phillips and Camille Bradford Fellowship.

** Research supported in part by the U.S. Army Research Laboratory.

K. Nyberg (Ed.): FSE 2008, LNCS 5086, pp. 303–315, 2008.

to generate a fresh set of uniformly random coins each time a message is encrypted. In practice, such coins might be generated by using a combination of randomness extractors and pseudorandom number generators (PRNGs) to distill pseudorandom coins from a high-entropy source available to the sender.

The above, however, neglects the possibility that the random coins used to encrypt may sometimes be "less than perfect". For example, the sender may be using a faulty PRNG that produces biased or partially predictable outputs. Or, the random source used to seed the PRNG may have less entropy than originally thought. More malicious scenarios include the possibilities that an adversary may have tampered with the PRNG used by the sender, or may be able to effect some control over the random source used to seed the PRNG. In the most extreme case, the adversary may have physical access to the device performing the encryption (as might be the case if, e.g., encryption is carried out on a lightweight device captured by the adversary), and may then have complete control over the "random coins" that will actually be used to encrypt. We refer to such attacks as *chosen-randomness attacks*.

In this work, we introduce new definitions of security that offer protection against the attacks just described. Our definitions assume the worst possible case: that the randomness used by the encryption oracle is under the complete control of the adversary. In fact, the only random coins that are not under the adversary's control (other than those used to generate the key) are those that are used to encrypt the challenge ciphertext; we assume those coins are truly random.[1] Our definition, then, can be viewed (informally) as offering *semantic security* for any messages that are encrypted using "good" random coins, even if the adversary is able to "probe" the system repeatedly and thereby cause the sender to use "poor" random coins when encrypting other messages.

Summary of our contributions. We formally define security against chosen-randomness attacks (CRA-security), both with and without the additional presence of a decryption oracle. We then show two secure constructions that can be based on any block cipher. The first is a relatively simple fixed-length construction, while the second is a scheme that can encrypt arbitrary-length messages. We also show a generic transformation converting any CPA-secure scheme into a CRA-secure scheme. Finally, we propose a simple way to extend any CRA-secure scheme so as to also achieve security against chosen-ciphertext attacks.

1.1 Related Work

The most relevant prior work is perhaps Rogaway's notion of *nonce-based* private-key encryption [15], which treats the encryption algorithm as a deterministic function of the message and a user-provided nonce. (With respect to

[1] It is not hard to see that *some* assumption regarding those coins is necessary in our setting (if the adversary has complete control over *all* coins, then the scheme degenerates to a deterministic one that cannot be secure); our assumption that the coins used to generate the challenge ciphertext are truly random is made for simplicity, and can be relaxed by using randomness extractors and assuming only access to a high-entropy source when encrypting the challenge ciphertext.

this viewpoint, it is the responsibility of the user — not the encryption algorithm — to ensure, e.g., that nonces are chosen at random.) In this context, Rogaway defines a notion of security that, roughly speaking, guarantees semantic security *as long as nonces never repeat.* While this definition is somewhat similar to our own, we show in Section 3.1 that the notion considered by Rogaway is *incomparable* to the notion of CRA-security considered here; i.e., there are schemes satisfying his definition and not ours, and vice versa. We remark further that the motivations for our work and Rogaway's work are very different: as argued by Rogaway [15], nonce-based security is best understood as a usability requirement, whereas we are interested in examining a stronger attack model (within the conventional framework for encryption).

Adversarial manipulation of a PRNG was mentioned as motivation for our work. While there has been prior work developing "forward-" and "backward-secure" pseudorandom number generators [4,1,8], simply composing such generators with a standard CPA- or CCA-secure encryption scheme does *not* defend against the attacks considered here. The reason is that these prior works consider only adversaries that *learn* the internal state of the PRNG, whereas our notions consider stronger adversaries that may *control* the state of the PRNG. One can therefore view our notion of CRA-security as achieving a strong variant of backward- and forward-security with respect to the underlying source of randomness. In other words, our definitions guarantee that a plaintext encrypted using high-quality randomness is protected even against adversaries that can control the source after the present plaintext is encrypted (i.e., strong forward-security), or that have controlled it in the past (i.e., strong backward-security).

Work of McInnes and Pinkas [13] and Dodis et al. [7,5] has also considered the security of encryption when truly random coins are not available to the sender. Although these works are superficially related to our own, the problems being considered — as well as the motivation — are very different. The work of [13,7,5] is unwilling to assume *any* truly random coins, even during generation of the secret key, and is interested in exploring what can be achieved in such an extreme setting. For this reason, they are primarily concerned with information-theoretic security (although later work [6,5] treats computational security) and do not consider security against chosen-plaintext attacks at all. In this work, in contrast, we are willing to assume that truly random coins *exist* (e.g., during key generation and, at least once, when encrypting), but are concerned that the adversary may periodically be able to tamper with the honest user's ability to generate true random coins. We are then interested in the question of whether the analogue of CPA-security is achievable.

2 Notation and Preliminaries

We use standard cryptographic notation and terminology. We use $\langle a, b\rangle$ or $a\|b$ interchangeably for the concatenation of strings a and b. We let $\mathsf{Func}[n, m]$ denote the set of all functions from $\{0,1\}^n$ to $\{0,1\}^m$, and let $\mathsf{Perm}[n, n]$ denote the set of all permutations over $\{0,1\}^n$. A function $f : \mathbb{N} \rightarrow \mathbb{N}$ is negligible in k, if for every polynomial p and sufficiently large k, $f(k) < 1/p(k)$. We write $\mathsf{negl}(k)$ and $\mathsf{poly}(k)$ to refer to unspecified negligible and polynomial functions in k.

If $\mathcal{O}$ is a probabilistic algorithm, then $\mathcal{O}(x)$ denotes an execution of $\mathcal{O}$ on input x with uniformly chosen random coins, and $\mathcal{O}(x;r)$ denotes an execution of $\mathcal{O}$ on input x with random coins r. Given a probabilistic algorithm $\mathcal{O}$, we will consider adversaries $\mathcal{A}$ given access to an oracle that on input $\langle x, r\rangle$ returns $\mathcal{O}(x;r)$ (this is different from the usual case where $\mathcal{A}$ is given access to an oracle that on input x returns $\mathcal{O}(x;r)$ for uniformly chosen random coins r).

2.1 Cryptographic Tools

We use standard cryptographic tools which are reviewed here to fix notation.

Pseudo-random functions. Let F be an efficiently-computable keyed function, where for a fixed key K of length k we have $F_K : \{0,1\}^{\ell_{in}(k)} \to \{0,1\}^{\ell_{out}(k)}$ with ℓ_{in}, ℓ_{out} polynomial in k. We say F is a *pseudorandom function* (PRF) if, informally, the function F_K (for a random key $K \in \{0,1\}^k$) is indistinguishable from a function f chosen uniformly at random from $\mathsf{Func}[\ell_{in}(k), \ell_{out}(k)]$. If F is a PRF and F_K is an efficiently-invertible permutation for each choice of K, then we call F a *pseudorandom permutation* (PRP). We refer to [11] for the formal definitions, which are standard.

Encryption. A private-key encryption scheme $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ consists of three polynomial-time algorithms with the following functionality:

- Gen takes as input a security parameter 1^k and returns a key K.
- Enc is a probabilistic algorithm that takes as input the key K and a message m from some associated message space. It returns a ciphertext c, and we denote this by $c \leftarrow \mathsf{Enc}_K(m)$.
- Dec takes as input the key K and a ciphertext c; it returns either a message m in the message space or a special failure symbol $\perp$. We write this as $m := \mathsf{Dec}_K(c)$, and assume without loss of generality that Dec is deterministic.

We assume perfect correctness: for all $k \in \mathbb{N}$, all K output by $\mathsf{Gen}(1^k)$, and all messages m in the message space, $\mathsf{Dec}_K(\mathsf{Enc}_K(m)) = m$.

In most schemes, $\mathsf{Gen}(1^k)$ simply outputs a random key of length k; when this is the case, we write $\mathsf{SKE} = (\mathsf{Enc}, \mathsf{Dec})$.

We use the standard notions of security against chosen-plaintext attacks (CPA-security) and security against (adaptive) chosen-ciphertext attacks (CCA-security); see, e.g., [11].

Message authentication codes (MACs). A message authentication code $\mathsf{MAC} = (\mathsf{Mac}, \mathsf{Vrfy})$ is a pair of polynomial-time algorithms. Mac takes as input a key $K \in \{0,1\}^k$ and a message $m \in \{0,1\}^*$ and outputs a tag t; we assume[2] that Mac is deterministic and denote this by $t := \mathsf{Mac}_K(m)$. The deterministic verification algorithm Vrfy takes as input a key $K \in \{0,1\}^k$, a message

[2] This is without loss of generality, and anyway holds for many common constructions.

$m \in \{0,1\}^*$, and a tag t; it outputs a bit $b := \mathsf{Vrfy}_K(m,t)$ where a '1' indicates acceptance and a '0' indicates rejection. We assume that for all $k \in \mathbb{N}$, all $K \in \{0,1\}^k$, and all $m \in \{0,1\}^*$ it holds that $\mathsf{Vrfy}_K(m, \mathsf{Mac}_K(m)) = 1$.

Note that we assume trivial key generation, where on security parameter 1^k the key is chosen uniformly from $\{0,1\}^k$. We also assume MACs for arbitrary-length messages. (Neither assumption is essential, but making these assumptions simplifies the presentation.)

We use the standard definition of existential unforgeability under an adaptive chosen message attack; see [11]. A MAC has *unique tags* if for all $k \in \mathbb{N}$, all $K \in \{0,1\}^k$, and all $m \in \{0,1\}^*$, there is a unique t such that $\mathsf{Vrfy}_K(m,t) = 1$.

3 Definitions

We now present our definitions of CRA and CCRA-security. Intuitively, CRA-security guarantees that, given a ciphertext, no polynomially-bounded adversary can recover any partial information about the plaintext even if it has access to an encryption oracle *and complete control over its source of randomness.*

Definition 1 (CRA-security). *Let* $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *be a private-key encryption scheme.* SKE *is* CRA-secure *if the advantage of any polynomial-time adversary* $\mathcal{A}$ *in the following experiment is negligible (in* k*):*

1. *First, a key* $K \leftarrow \mathsf{Gen}(1^k)$ *is generated.*
2. $\mathcal{A}$ *is allowed to interact with an oracle* $\mathsf{Enc}_K(\cdot\,;\,\cdot)$*. We stress that, here,* $\mathcal{A}$ *submits pairs* $\langle m, r\rangle$ *and, in return, is given* $\mathsf{Enc}_K(m; r)$*. (Since* $\mathcal{A}$ *can choose* r *uniformly at random, this is at least as strong as a chosen-plaintext attack.)*
3. $\mathcal{A}$ *outputs two equal-length messages* m_0, m_1*. A bit* b *is chosen at random, and a "challenge ciphertext"* $c \leftarrow \mathsf{Enc}_K(m_b)$ *is computed and given to* $\mathcal{A}$*. We stress that encryption here uses uniform coins that are not known to* $\mathcal{A}$*.*
4. $\mathcal{A}$ *may continue to interact with its oracle as before. Eventually, it outputs a bit* b'*; the experiment evaluates to 1 if* $b' = b$*.*

We denote the above experiment by $\mathsf{CRA}_{\mathcal{A},\mathsf{SKE}}(k)$*, and define the advantage of* $\mathcal{A}$ *in the experiment as* $\left|\Pr[\mathsf{CRA}_{\mathcal{A},\mathsf{SKE}}(k) = 1] - \frac{1}{2}\right|$.

The stronger notion of CCRA-security guarantees that, given a ciphertext, no polynomially-bounded adversary can recover any partial information about the plaintext, even if it has access to both an encryption and a decryption oracle and complete control over the encryption oracle's source of randomness. This is defined in the natural way, and we denote the experiment in this case by $\mathsf{CCRA}_{\mathcal{A},\mathsf{SKE}}(k)$. We remark that since Dec is deterministic, there is no analogue of the adversary's being able to "control the randomness" used during decryption.

3.1 Comparison to Previous Definitions

In this section we compare our new definitions to previous security notions for private-key encryption, including CPA- and CCA-security, and the more closely related notions of nonce-based CPA and CCA-security from [15].

Theorem 1. *CRA-security is strictly stronger than CPA-security.*

Proof. It is easy to see that CRA-security implies CPA-security. We show that the converse is not true. Let F be a pseudorandom function, and consider the standard CPA-secure private-key encryption scheme with encryption given by $\mathsf{Enc}_K(m; r) = \langle r, F_K(r) \oplus m \rangle$. We claim that this scheme is not CRA-secure. To see this, note that an adversary given a challenge ciphertext $\langle r, c \rangle$ can submit $\langle 0^k, r \rangle$ to its oracle and will receive in return the ciphertext

$$\mathsf{Enc}_K(0^k; r) = \langle r, F_K(r) \rangle.$$

It is then trivial for the adversary to determine the message that was encrypted.

Theorem 2. *CCRA-security is strictly stronger than CCA-security.*

A proof is very similar to the proof of the previous theorem, and is omitted.

Nonce-based encryption [15] is a formalization of private-key encryption where the encryption algorithm is a deterministic function of a message and a nonce, and the user (or, more generally, the program calling the encryption algorithm as a sub-routine) is responsible for providing the nonce. E.g., in the case of CBC-mode encryption the IV would be an additional input provided to the encryption algorithm as opposed to being generated "internally". This formulation gives more flexibility with respect to how the nonce is chosen: by assuming the nonce is chosen uniformly each time the encryption algorithm is called, the standard notion of probabilistic encryption is recovered, but another option is to assume only that nonces never repeat (but are not necessarily random).

Rogaway [15] considers definitions of security for nonce-based schemes in which the adversary is given some control over the nonce that is used to encrypt *at all times*, i.e., both when interacting with an encryption oracle as well as when the challenge ciphertext is computed. (A definition of nonce-based CPA-security is given in Appendix A.) Intuitively, these definitions are incomparable to our own because:

– On one hand, we assume the adversary has *no control* over the randomness used to encrypt the challenge ciphertext, whereas Rogaway allows the adversary to have some control over the randomness even in this case.
– On the other hand, we give the adversary *full control* over the randomness used by the encryption oracle, whereas Rogaway restricts the adversary to never using the same nonce twice.

We formally prove that the notions are incomparable now.

Theorem 3. *Nonce-based CPA-security and CRA-security are incomparable.*

The theorem is a consequence of the following two lemmas. In proving them, we rely on the definition of nonce-based security given in Appendix A; the definition is weaker than that given in [15], but the difference is inessential and unimportant for the present discussion.

Lemma 1. *Assuming the existence of one-way functions, there is a private-key encryption scheme that is nonce-based CPA-secure but that is not CRA-secure.*

Proof. We take the standard encryption scheme used in the proof of Theorem 1. Recall, F is a pseudorandom function, which may be constructed from any one-way function. Encryption is given by $\mathsf{Enc}_K(m;r) = \langle r, F_K(r) \oplus m \rangle$, where we treat r as a nonce, and decryption is given by $\mathsf{Dec}_K(\langle r, c \rangle) = F_K(r) \oplus c$.

We have already shown in the proof of Theorem 1 that this scheme is not CRA-secure. On the other hand, it is not hard to see that it is nonce-based CPA-secure: since the adversary is not allowed to use the same nonce twice, it holds in particular that the nonce r used when encrypting the challenge ciphertext is distinct from any nonce used in answering any queries to the encryption oracle. It then follows easily from the pseudorandomness of F that the scheme is nonce-based CPA-secure.

Lemma 2. *Assuming the existence of a private-key encryption scheme that is CRA-secure, there is a CRA-secure scheme that is not nonce-based CPA-secure.*

Proof. Let $\mathsf{SKE} = (\mathsf{Enc}, \mathsf{Dec})$ be a CRA-secure private-key encryption scheme. Assume without loss of generality that, on security parameter k, the encryption algorithm uses k bits of randomness. (We will again treat the random coins used by Enc as a nonce.) Define a modified encryption scheme $\mathsf{SKE}' = (\mathsf{Enc}', \mathsf{Dec})$ (decryption remains unchanged) as follows:

$$\mathsf{Enc}'_K(m; r\|b) = \mathsf{Enc}_K(m; r),$$

where b is a bit and $r \in \{0,1\}^k$. It is easy to see that SKE' is not nonce-based CPA secure: an adversary can simply request to have the challenge ciphertext encrypted using the nonce $r\|0$ and then query its encryption oracle using the (distinct) nonce $r\|1$. (Further details omitted.) It is similarly easy to see that SKE' remains CRA-secure: oracle queries with respect to the modified scheme SKE' are no more powerful than oracle queries with respect to the original scheme SKE; when the challenge ciphertext is encrypted, it will be encrypted using algorithm Enc with uniform random coins.

In the sections that follow, we will show constructions of CRA-secure encryption schemes that may be based on any one-way function.

Using ideas as above, we can similarly show that the notions of CCRA-security and nonce-based CCA-security are incomparable.

4 Achieving CRA-Security

In this section we propose two CRA-secure private-key encryption schemes based on PRPs; our first construction handles fixed-length messages only, while our second construction handles messages of variable length. We then show a general transformation from any CPA-secure scheme to a CRA-secure one.

4.1 A Fixed-Length CRA-Secure Construction

Our first construction is a modification of the standard CPA-secure encryption scheme that we have seen before in the proof of Theorem [1]. Let P be a pseudorandom *permutation* on k-bit strings, and let F be a pseudorandom function mapping k-bit inputs to k-bit outputs. Our scheme is defined as follows:

$\mathsf{Gen}(1^k)$: Choose $K_1, K_2 \leftarrow \{0,1\}^k$ and output $K = \langle K_1, K_2 \rangle$.

$\mathsf{Enc}_K(m;r)$: Compute $c_2 = F_{K_2}(r) \oplus m$, then output the ciphertext $\langle P_{K_1}(r), c_2 \rangle$.

$\mathsf{Dec}_K(\langle c_1, c_2 \rangle)$: Compute $r = P^{-1}_{K_1}(c_1)$. Then output $m := F_{K_2}(r) \oplus c_2$ as the message.

Theorem 4. *If P is a pseudorandom permutation and F is a pseudorandom function, then the scheme described above is CRA-secure.*

Proof. Consider the (imaginary) scheme $\widetilde{\mathsf{SKE}} = (\widetilde{\mathsf{Gen}}, \widetilde{\mathsf{Enc}}, \widetilde{\mathsf{Dec}})$ in which $\widetilde{\mathsf{Gen}}$ samples $p \leftarrow \mathsf{Perm}[k,k]$ and $f \leftarrow \mathsf{Func}[k,k]$ uniformly at random, and $\widetilde{\mathsf{Enc}}(m;r)$ outputs the ciphertext $\langle p(r), f(r) \oplus m \rangle$. We analyze the security of this scheme in an information-theoretic sense; CRA-security of the scheme described above (for polynomial-time adversaries) then follows easily.

Let $\mathcal{A}$ be an adversary making at most $q(k)$ queries to its oracle in experiment $\mathsf{CRA}_{\mathcal{A},\widetilde{\mathsf{SKE}}}(k)$. Let r be the randomness used to generate the challenge ciphertext in this experiment, and let query be the event that one of $\mathcal{A}$'s oracle queries uses randomness r. Then:

$$\begin{aligned}\Pr\left[\mathsf{CRA}_{\mathcal{A},\widetilde{\mathsf{SKE}}}(k) = 1\right] &= \Pr\left[b' = b\right] \\ &= \Pr\left[b' = b \wedge \mathsf{query}\right] + \Pr\left[b' = b \wedge \overline{\mathsf{query}}\right]\end{aligned}$$

from which it follows that

$$\left|\Pr\left[\mathsf{CRA}_{\mathcal{A},\widetilde{\mathsf{SKE}}}(k) = 1\right] - \Pr\left[b' = b \mid \overline{\mathsf{query}}\right]\right| \leq \Pr\left[\mathsf{query}\right].$$

The following two claims complete the proof of the theorem.

Claim. $\Pr\left[\mathsf{query}\right] \leq q(k)/2^k$.

This follows from the fact that, after observing the challenge ciphertext and making q' queries to its oracle that do not use randomness r, all $\mathcal{A}$ knows is that r is not equal to any of the random coins used in its queries thus far.

Claim. $\Pr[b' = b \mid \overline{\mathsf{query}}] = 1/2$.

Let m_0, m_1 denote the messages output by $\mathcal{A}$, and let $\langle c_1, c_2 \rangle$ be the challenge ciphertext. If query does not occur, then $f(r)$ is equally likely to be $c_2 \oplus m_0$ or $c_2 \oplus m_1$ (just as in the one-time pad), and thus $\mathcal{A}$ can do no better than guess.

4.2 A CRA-Secure Construction for Variable-Length Messages

Our second construction applies a similar modification as in the previous section, but to CTR-mode encryption. Let P be a pseudorandom permutation and F a pseudorandom function, as in the previous section.

$\mathsf{Gen}(1^k)$: Choose $K_1, K_2 \leftarrow \{0,1\}^k$ and output $K = \langle K_1, K_2 \rangle$.

$\mathsf{Enc}_K(m; r)$: Parse m into ℓ blocks $m = \langle m_1, \ldots, m_\ell \rangle$, each of length k. For $1 \le i \le \ell$, compute $c_i = F_{K_2}(r+i) \oplus m_i$. (Here, we are viewing r as a k-bit integer and addition is done modulo 2^k). Output the ciphertext $c = \langle P_{K_1}(r), c_1, \ldots, c_\ell \rangle$.

$\mathsf{Dec}_K(\langle c_0, c_1, \ldots, c_\ell \rangle)$: Compute $r = P^{-1}_{K_1}(c_0)$. For $1 \le i \le \ell$, compute $m_i = F_{K_2}(r+i) \oplus c_i$. Output $m = \langle m_1, \ldots, m_\ell \rangle$.

Theorem 5. *If P is a pseudorandom permutation and F is a pseudorandom function then the scheme described above is CRA-secure.*

Proof. Consider the private-key encryption scheme $(\widetilde{\mathsf{Gen}}, \widetilde{\mathsf{Enc}}, \widetilde{\mathsf{Dec}})$ such that $\widetilde{\mathsf{Gen}}$ samples $p \leftarrow \mathsf{Perm}[k,k]$ and $f \leftarrow \mathsf{Func}[k,k]$ uniformly at random, and then $\widetilde{\mathsf{Enc}}$ is define in the natural way based on the scheme described above. We analyze the security of this scheme in an information-theoretic sense; security of the scheme described above (for polynomial-time adversaries) then follows easily.

Let $\mathcal{A}$ be an adversary making at most $q = q(k)$ queries to its oracle in experiment $\mathsf{CRA}_{\mathcal{A},\widetilde{\mathsf{SKE}}}(k)$, where the messages in these queries have block-length at most $\ell = q(k)$. We also let $q(k)$ be a bound on the block-length of the messages m_0, m_1 output by $\mathcal{A}$. Let r be the randomness used to generate the challenge ciphertext in this experiment, and let query be the event that one of $\mathcal{A}$'s oracle queries uses randomness $r' \in \{r-q+1, \ldots, r+q-1\}$. Then:

$$\begin{aligned}\Pr\left[\mathsf{CRA}_{\mathcal{A},\widetilde{\mathsf{SKE}}}(k) = 1\right] &= \Pr\left[b' = b\right] \\ &= \Pr\left[b' = b \wedge \mathsf{query}\right] + \Pr\left[b' = b \wedge \overline{\mathsf{query}}\right]\end{aligned}$$

from which it follows that

$$\left|\Pr\left[\mathsf{CRA}_{\mathcal{A},\widetilde{\mathsf{SKE}}}(k) = 1\right] - \Pr\left[b' = b \mid \overline{\mathsf{query}}\right]\right| \le \Pr\left[\mathsf{query}\right].$$

The following two claims complete the proof of the theorem.

Claim. $\Pr\left[\mathsf{query}\right] \le \mathcal{O}(q(k)^2/2^k)$.

Intuitively, the value r used to encrypt the challenge ciphertext is "hidden" from $\mathcal{A}$. Thus, assuming query has not yet occurred, a query made by $\mathcal{A}$ to its encryption oracle can cause query to occur with probability at most

$$\frac{(r+q-1) - (r-q+1) + 1}{2^k} = \frac{2q-1}{2^k}.$$

Applying a union bound to the q queries of $\mathcal{A}$ gives the stated result.

Claim. $\Pr[b' = b \mid \overline{\mathsf{query}}] = 1/2$.

This follows by analogy to the one-time pad; conditioned on query not occurring, $F_{K_2}(r+1), \ldots, F_{K_2}(r+q)$ are uniformly distributed from $\mathcal{A}$'s point of view.

4.3 A CPA-to-CRA Transformation

Finally, we present a transformation that turns any CPA-secure private-key encryption scheme into a CRA-secure scheme. The transformation assumes the existence of pseudorandom functions for arbitrary-length inputs; these may be constructed based on any one-way function, whose existence is implied by the existence of a CPA-secure encryption scheme.

Let $\mathsf{SKE}' = (\mathsf{Gen}', \mathsf{Enc}', \mathsf{Dec}')$ be a CPA-secure encryption scheme in which encryption uses k random coins (this is not essential, but makes the analysis easier), and let F be a pseudorandom function. Define SKE as follows:

$\mathsf{Gen}(1^k)$: Compute $K_1 \leftarrow \mathsf{Gen}'(1^k)$, and then choose $K_2 \leftarrow \{0,1\}^k$. Output the key $K = \langle K_1, K_2 \rangle$.

$\mathsf{Enc}_K(m; r)$, where $r \in \{0,1\}^k$: Compute "random coins" $r' = F_{K_2}(m \| r)$. Then output the ciphertext $c' = \mathsf{Enc}'_{K_1}(m; r')$.

$\mathsf{Dec}_K(c')$: Output $m = \mathsf{Dec}'_{K_1}(c')$.

Theorem 6. *If* SKE' *is a CPA-secure private-key encryption scheme and* F *is a pseudorandom function, then the scheme described above is CRA-secure.*

Proof. Given an adversary $\mathcal{A}$ attacking the constructed scheme (in the sense of CRA-security), we construct an adversary $\mathcal{A}'$ attacking SKE' (in the sense of CPA-security). Our adversary $\mathcal{A}'$ is defined as follows:

1. Run $\mathcal{A}$. When $\mathcal{A}$ makes oracle query $\langle m, r \rangle$ to its oracle, $\mathcal{A}'$ queries m to its own (standard) encryption oracle and returns the result to $\mathcal{A}$. We assume without loss of generality that $\mathcal{A}$ never makes the same oracle query twice.
2. When $\mathcal{A}$ outputs two messages m_0, m_1, these same messages are output by $\mathcal{A}'$. The challenge ciphertext given to $\mathcal{A}'$ is forwarded to $\mathcal{A}$.
3. Oracle queries of $\mathcal{A}$ are handled exactly as before.
4. When $\mathcal{A}$ outputs a bit b', the same bit is output by $\mathcal{A}'$.

It is not hard to see that the view of $\mathcal{A}$ in the above is computationally indistinguishable from its view when attacking the constructed scheme. Thus, the advantage of $\mathcal{A}'$ is negligibly close to the advantage of $\mathcal{A}$. Since SKE' is CPA-secure by assumption, we conclude that the advantage of $\mathcal{A}$ in attacking the constructed scheme (in the sense of CRA-security) is negligible.

5 Achieving CCRA-Security

We now show that the standard "encrypt-then-MAC" transformation [3] from CPA-secure schemes to CCA-secure ones works in our setting also. Let $(\mathsf{Mac}, \mathsf{Vrfy})$ be a secure message authentication code, and let $\mathsf{SKE}' = (\mathsf{Gen}', \mathsf{Enc}', \mathsf{Dec}')$ be a CRA-secure encryption scheme. Define SKE as follows:

$\mathsf{Gen}(1^k)$: Compute $K_1 \leftarrow \mathsf{Gen}'(1^k)$, and then choose $K_2 \leftarrow \{0,1\}^k$. Output the key $K = \langle K_1, K_2 \rangle$.

$\mathsf{Enc}_K(m; r)$: Compute $c' = \mathsf{Enc}'_{K_1}(m; r)$ and $t = \mathsf{Mac}_{K_2}(c')$. Output the ciphertext $\langle c', t \rangle$.

$\mathsf{Dec}_K(\langle c', t\rangle)$: If $\mathsf{Vrfy}_{K_2}(c', t) = 1$ then output $\mathsf{Dec}'_{K_1}(c')$. Otherwise output $\perp$.

Theorem 7. *If* SKE' *is CRA-secure and* $(\mathsf{Mac}, \mathsf{Vrfy})$ *is a secure MAC with unique tags, then the scheme described above is CCRA-secure.*

Proof. Let $\mathcal{A}$ be an adversary attacking SKE in the sense of CCRA-security. Let query be the event that $\mathcal{A}$ submits a decryption query $\langle c, t\rangle$ to its decryption oracle such that $\mathsf{Dec}_K(c, t) \neq \perp$ and $\langle c, t\rangle$ was not the result of a previous encryption query. Clearly,

$$\Pr[\,b' = b\,] = \Pr[\,b' = b \wedge \mathsf{query}\,] + \Pr[\,b' = b \wedge \overline{\mathsf{query}}\,]$$

from which it follows that

$$|\Pr[\,b' = b\,] - \Pr[b' = b \mid \overline{\mathsf{query}}]| \leq \Pr[\,\mathsf{query}\,].$$

The following claims complete the proof.

Claim. For all PPT adversaries $\mathcal{A}$ it holds that $\Pr[\,\mathsf{query}\,]$ is negligible.

We show that if there exists a PPT adversary $\mathcal{A}$ such that $\Pr[\,\mathsf{query}\,]$ is not negligible, then there exists a PPT adversary $\mathcal{B}$ that can win the existential unforgeability experiment against MAC with non-negligible probability.

Consider the adversary $\mathcal{B}$ that, given 1^k and oracle access to $\mathsf{Mac}_K(\cdot)$ and $\mathsf{Vrfy}_K(\cdot,\cdot)$, begins by generating an encryption key $K_1 \leftarrow \mathsf{Gen}'(1^k)$ and runs $\mathcal{A}(1^k)$ as follows,

> Given an encryption query $e = \langle m, r\rangle$, adversary $\mathcal{B}$ computes $c \leftarrow \mathsf{Enc}'_{K_1}(m; r)$ and queries its own Mac oracle with c, receiving t. Finally, it returns the ciphertext $\langle c, t\rangle$ to $\mathcal{A}$.
>
> Given a decryption query $d = \langle c, t\rangle$, adversary $\mathcal{B}$ queries its Vrfy oracle with c and t. If the oracle returns 1 then it computes and returns $m \leftarrow \mathsf{Dec}'_{K_1}(c)$ to $\mathcal{A}$; otherwise it returns $\perp$. Adversary $\mathcal{B}$ stores all of $\mathcal{A}$'s decryption queries. After polynomially-many queries, $\mathcal{A}$ outputs m_0, m_1.

$\mathcal{B}$ samples $b \leftarrow \{0, 1\}$, computes $c^* \leftarrow \mathsf{Enc}'_{K_1}(m_b)$, and queries its oracle to receive $t^* \leftarrow \mathsf{Mac}_K(c^*)$. It then runs $\mathcal{A}$ with the challenge ciphertext $\langle c^*, t^*\rangle$, and answers its queries as before. After polynomially many queries, $\mathcal{A}$ outputs a bit b' and halts. Let $q(k)$ be the number of decryption queries made by $\mathcal{A}$. If query has occurred by the end of the game (note that $\mathcal{B}$ can determine if this happens), then $\mathcal{B}$ outputs the appropriate query for which this first occurred.

Notice that $\mathcal{B}$ succeeds if query occurs. Since $\mathcal{A}$'s view is identical to its view when attacking SKE, the claim follows.

Claim. For all PPT adversaries $\mathcal{A}$, it holds that $\Pr[b' = b \mid \overline{\mathsf{query}}] \leq 1/2 + \mathsf{negl}(k)$.

We show that if there exists a PPT adversary $\mathcal{A}$ such that

$$\Pr[b' = b \mid \overline{\mathsf{query}}] \geq 1/2 + 1/\mathsf{poly}(k),$$

then there exists a PPT adversary $\mathcal{B}$ that can succeed in attacking SKE' (in the sense of CRA-security) with non-negligible probability.

Consider $\mathcal{B}$ that, given 1^k, begins by choosing $K_2 \in \{0,1\}^k$ and runs $\mathcal{A}_1(1^k)$ as follows.

> Given an encryption query $\langle m, r\rangle$, adversary $\mathcal{B}$ queries its oracle with $\langle m, r\rangle$ to obtain a ciphertext c. It then computes $t \leftarrow \mathsf{Mac}_{K_2}(c)$, and returns the ciphertext $\langle c, t\rangle$ to $\mathcal{A}$. It stores the tuple $\langle c, t, m\rangle$ in a table T.
>
> Given a decryption query $d = \langle c, t\rangle$, adversary $\mathcal{B}$ looks up the pair $\langle c, t\rangle$ in its table and returns the corresponding plaintext m. If the pair $\langle c, t\rangle$ is not in T, then it returns $\perp$. After polynomially many queries, $\mathcal{A}$ outputs messages m_0, m_1 which $\mathcal{B}$ also outputs.

Given a challenge ciphertext c^*, adversary $\mathcal{B}$ computes $t^* \leftarrow \mathsf{Mac}_{K_2}(c^*)$ and gives the challenge ciphertext $\langle c^*, t^*\rangle$ to $\mathcal{A}$, answering its oracle queries as before. Eventually, $\mathcal{A}$ outputs a bit b' which $\mathcal{B}$ outputs as well.

It remains to analyze $\mathcal{B}$'s success probability. First, notice that $\mathcal{B}$ can answer $\mathcal{A}$'s encryption queries perfectly. Furthermore, if query does not occur, then the only valid decryption queries $\mathcal{A}$ makes are for ciphertexts that were the result of previous encryption queries. In this case (i.e., conditioned on $\overline{\mathsf{query}}$), $\mathcal{B}$ will also correctly answer all of $\mathcal{A}$'s decryption queries (using its table). It follows then that conditioned on $\overline{\mathsf{query}}$, the view of $\mathcal{A}$ is identical to its view when attacking SKE. The claim follows.

References

1. Barak, B., Halevi, S.: A model and architecture for pseudorandom generation and applications to /dev/random. In: ACM Conf. on Computer and Communications Security (2005)
2. Bellare, M., Desai, A., Jokipii, E., Rogaway, P.: A concrete security treatment of symmetric encryption. In: 38th Annual Symposium on Foundations of Computer Science (FOCS) (1997)
3. Bellare, M., Namprempre, C.: Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In: Okamoto, T. (ed.) ASIACRYPT 2000. LNCS, vol. 1976. Springer, Heidelberg (2000)
4. Bellare, M., Yee, B.: Forward-security in private-key cryptography. In: RSA Cryptographers' Track 2003 (2003)
5. Bosley, C., Dodis, Y.: Does privacy require true randomness? In: Theory of Cryptography Conference 2007 (2007)
6. Dodis, Y., Ong, S.J., Prabhakaran, M., Sahai, A.: On the (im)possibility of cryptography with imperfect randomness. In: 45th Annual Symposium on Foundations of Computer Science (FOCS) (2004)
7. Dodis, Y., Spencer, J.: On the (non)universality of the one-time pad. In: 43rd Annual Symposium on Foundations of Computer Science (FOCS) (2002)
8. Fu, K., Kamara, S., Kohno, T.: Key regression: Enabling efficient key distribution for secure distributed storage. In: NDSS 2006 (2006)

9. Goldreich, O., Goldwasser, S., Micali, S.: How to construct random functions. J. ACM 33(4), 792–807 (1984)
10. Goldwasser, S., Micali, S.: Probabilistic encryption. Journal of Computer and System Sciences 28(2), 270–299 (1984)
11. Katz, J., Lindell, Y.: Introduction to Modern Cryptography. Chapman & Hall/CRC Press (2007)
12. Katz, J., Yung, M.: Characterization of security notions for probabilistic private-key encryption. J. Cryptology 19(1), 67–96 (2006)
13. McInnes, J., Pinkas, B.: On the impossibility of private-key cryptography with weakly random keys. In: Menezes, A., Vanstone, S.A. (eds.) CRYPTO 1990. LNCS, vol. 537. Springer, Heidelberg (1991)
14. Rackoff, C., Simon, D.: Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. In: Feigenbaum, J. (ed.) CRYPTO 1991. LNCS, vol. 576. Springer, Heidelberg (1992)
15. Rogaway, P.: Nonce-based symmetric encryption. In: Roy, B., Meier, W. (eds.) FSE 2004. LNCS, vol. 3017. Springer, Heidelberg (2004)

A Nonce-Based Private-Key Encryption

We offer a definition in the spirit of nonce-based security [15], but we do not require that ciphertexts be indistinguishable from random strings. (This extra requirement is irrelevant as far as the results of the present paper are concerned.)

Definition 2 (Nonce-based CPA-security). *Let* $\mathsf{SKE} = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *be a private-key encryption scheme where encryption uses* k *random coins and we treat these coins as a nonce.* SKE *is* nonce-based CPA-secure *if the advantage of any polynomial-time adversary* $\mathcal{A}$ *in the following experiment is negligible (in* k*):*

1. *First, a key* $K \leftarrow \mathsf{Gen}(1^k)$ *is generated. Set* $\mathsf{Nonces} = \{0,1\}^k$.
2. $\mathcal{A}$ *is allowed to adaptively submit multiple queries of the form* $\langle m, r\rangle$*, subject always to the restriction that* $r \in \mathsf{Nonces}$. *In response to each such a query,* $\mathcal{A}$ *is given* $c = \mathsf{Enc}_K(m; r)$ *and* r *is removed from* Nonces.
3. $\mathcal{A}$ *outputs two equal-length messages* m_0, m_1 *and a nonce* $r \in \mathsf{Nonces}$. *A bit* b *is chosen at random, and a "challenge ciphertext"* $c = \mathsf{Enc}_K(m_b; r)$ *is computed and given to* $\mathcal{A}$*. Also,* r *is removed from* Nonces.
4. $\mathcal{A}$ *may continue to interact with its oracle as before. Eventually, it outputs a bit* b'*; the experiment evaluates to 1 if* $b' = b$.

We denote the above experiment by $\mathsf{NB\text{-}CPA}_{\mathcal{A},\mathsf{SKE}}(k)$*, and define the advantage of* $\mathcal{A}$ *in the experiment as* $\left|\Pr[\mathsf{NB\text{-}CPA}_{\mathcal{A},\mathsf{SKE}}(k) = 1] - \frac{1}{2}\right|$.

A One-Pass Mode of Operation for Deterministic Message Authentication—Security beyond the Birthday Barrier

Kan Yasuda

NTT Information Sharing Platform Laboratories, NTT Corporation
3-9-11 Midoricho Musashino-shi, Tokyo 180-8585 Japan
yasuda.kan@lab.ntt.co.jp

Abstract. We present a novel mode of operation which iterates a compression function $f : \{0,1\}^{n+b} \to \{0,1\}^n$ meeting a condition $b \geq 2n$. Our construction can be viewed as a way of domain extension, applicable to a fixed-input-length PRF (pseudo-random function) $f_k : \{0,1\}^b \to \{0,1\}^n$ meeting the condition $b \geq 2n$, which yields an arbitrary-input-length PRF $F_k : \{0,1\}^* \to \{0,1\}^n$. Our construction accomplishes both high security (beyond the birthday barrier) and high efficiency (one-pass), with engineering considerations of being stateless, deterministic and single-keyed.

Keywords: pseudo-random function, domain extension, birthday barrier, compression function, mode of operation, message authentication code, tweak, checksum, quasi-random function.

1 Introduction

Birthday Barrier. A message authentication code (MAC) is often constructed of a compression function (*e.g.*, HMAC [1]) via a mode of operation or a block cipher (*e.g.*, CBC-MAC [2]). The security of HMAC and CBC-MAC is based on the fact that they are pseudo-random functions (PRFs), assuming that the underlying primitives (*i.e.*, the compression function and the block cipher) are PRFs. Unfortunately, HMAC and CBC-MAC are inherently vulnerable to birthday attacks due to their naively-chained internal structure [3,4]. That is, using an n-bit-output compression function or block cipher, HMAC or CBC-MAC gets forged after about $2^{n/2}$ (which is much smaller than the desired 2^n) queries. This generic principle is known as the *birthday barrier*.

For modern compression functions and block ciphers the above attacks require, for example, 2^{128} and 2^{64} queries, which are unlikely to be a practical threat in most scenarios. It is rather a theoretical challenge to construct a mode with security beyond the birthday barrier at minimal costs over existing modes of operation.

Two Already-Known Ways of Breaking the Barrier. It seems that there exist roughly two approaches of breaking the barrier, and hence constructing MACs whose security is beyond the birthday bound. One is to allow use of

K. Nyberg (Ed.): FSE 2008, LNCS 5086, pp. 316–333, 2008.

either nonce elements or random salts. The other is to allow use of multiple passes. Yet, neither of these two approaches is satisfactory, as explained below.

Nonce elements are often used in encryption (*e.g.*, stream ciphers, the counter mode of block ciphers, *etc.*), but their presence is sometimes unwelcome in practical MAC applications; if a nonce value is used in a MAC scheme, then the value needs to be communicated, synchronized and maintained among all parties generating tags and/or verifying message-tag pairs. If instead a random salt is used, then these constraints become somewhat relaxed, but it still leaves problematic properties: the tag size gets enlarged, and the parties creating tags are required to possess a random-number generator.

The use of multiple passes offers construction without counters nor coins but results in inefficiency. Although usually parallelizable owing to their multi-pass structure, these schemes require more numbers of invocations to the underlying primitive, and the performance advantage due to the parallelism depends on each implementation and is generally limited.

Our Contributions. In this paper we devise a novel approach of breaking the birthday barrier. Namely, we utilize some techniques from the area of tweakable block ciphers and combine them with "checksum construction." The combination enables us to provide a one-pass mode of operation that overcomes the birthday limit without relying on the use of counters or coins.

Our starting primitive (*i.e.*, building block) is a compression function $f : \{0,1\}^{n+b} \to \{0,1\}^n$. We require that $b \geq 2n$. We emphasize that this requirement is essential in our construction; we utilize this condition in two (completely) different places.[1] Then using this primitive f, we construct a PRF $F_k : \{0,1\}^* \to \{0,1\}^n$ that satisfies the following seven properties:

1. The security of F is beyond the birthday barrier,
2. F is one-pass, that is, to process a message $M \in \{0,1\}^*$ only requires $|M|/b$ plus a small constant number of invocations to f,
3. Workings outside f consist of only simple machine operations,
4. F is stateless, avoiding use of nonce values or counters,
5. F is deterministic, avoiding use of random salts,
6. F is single-keyed, invoking f only with a fixed key $k \in \{0,1\}^n$ via $f_k(m) \overset{\text{def}}{=} f(k\|m)$ for a message block $m \in \{0,1\}^b$, and
7. The security of F is based on the sole assumption that f_k is a PRF.

It appears that no prior mode of operation, iterating either a compression function $f : \{0,1\}^{n+b} \to \{0,1\}^n$ or a block cipher $f_k : \{0,1\}^n \to \{0,1\}^n$, accomplishes the above features concurrently.

Organization of the Paper. Section 2 goes through previous work in this field. We then review necessary notions from the area of tweakable block ciphers

[1] We remark that the condition $b \geq 2n$ is not severe limitation in practice. In fact, off-the-shelf compression functions, such as $\mathsf{sha1} : \{0,1\}^{160+512} \to \{0,1\}^{160}$ and $\mathsf{sha256} : \{0,1\}^{256+512} \to \{0,1\}^{256}$, satisfy this requirement.

in Sect. 3. We introduce our mode of operation in Sect. 4. The security proofs of our mode are given in Sect. 5. A couple of techniques to improve the performance of our mode are discussed in Sect. 6. We mention some open problems regarding the domain extension of PRFs in Sect. 7, prior to concluding the paper in Sect. 8.

2 Previous Work

In this section we briefly look over previous constructs that break the birthday barrier, including the ones that take the two approaches mentioned in Sect. 1. Other known results, which have some relevance to the techniques used in the paper, are also cited in Sect. 3, 7 and 8.

Stateful or Randomized Construction. XOR MAC [5] is a parallelizable MAC that is based on a compression function. RMAC [6] is a serial MAC that is based on a block cipher. Both of these MACs guarantee security beyond the birthday barrier, yet XOR MAC is nonce-based and RMAC is a randomized algorithm.

Multiple-Pass-Based Construction. The idea of using two (or more) passes of data processing dates back to the design of RIPEMD and its application to Two-Track MAC [7]. A similar approach appears in the context of keyless hash functions as “Double-Pipe” hash [8]. These constructs effectively preclude birthday attacks, but the problem is that they are twice or more slower than their “single-pass” versions (even though they are somewhat parallelizable). The L-Lane scheme [9] performs better than a naively-doubled construction, but it is still less efficient as compared to a truly-single-pass construction.

Universal-Hash-Based Construction. A similar situation applies to MACs based on universal hash functions. UMAC [10] and MACRX [11] achieve security beyond the birthday barrier, but UMAC is nonce-based and MACRX is randomized. Once these MACs are made deterministic (in the obvious way), the security of such MACs gets degraded behind the birthday barrier immediately.

“Wide-Pipe” and Others. If we used a “wide-pipe” compression function $f : \{0,1\}^{2n+b} \rightarrow \{0,1\}^{2n}$ or a “wide” universal hashing with a collision probability $\varepsilon \approx 2^{-2n}$ (in a deterministic MAC), then we could certainly preclude the birthday attacks (in a provably secure way). However, such a method does not solve our problem at hand in nature; such a function deserves $2n$-bit security, not n-bit, or not to mention the fact that schemes based on wide functions would become inefficient.

Yet another approach is to construct a PRF $f'_{k'} : \{0,1\}^{2n} \rightarrow \{0,1\}^{2n}$ from a PRF $f_k : \{0,1\}^n \rightarrow \{0,1\}^n$ in a birthday-resistant way. Examples include Benes [12], Ω_t [13] and Feistel-6 [14]. These constructs however require too many (4 or more) invocations to f, and consequently schemes based on such an f' would be inefficient.

Lastly, we mention the Sum construction [15] which gives a way to construct a PRF from PRPs. The security of the resulting PRF is shown to be beyond the birthday limit, but the construction requires at least two invocations to f when instantiated with a single PRP f.

3 Preliminaries

In this section we first review the notion of pseudo-random functions (PRFs) and that of quasi-random functions (QRFs). We then give an overview of the theory of tweakable PRFs. Notice that such a theory is usually based on the framework of block ciphers, but we carefully restate the theory in the language of compression functions (rather than block ciphers). Some parts of the theory are directly translated into the new setting, while other parts need to receive local treatment in the context of compression functions.

Pseudo-random Functions (PRFs). Let $\{f_k : X \to Y\}$ be a family of functions with keys $k \in K$. Informally, we say that f is *pseudo-random* if f_k with a key $k \xleftarrow{\$} K$ randomly chosen is indistinguishable from a truly random function $\varphi : X \to Y$ (*i.e.*, $\varphi \xleftarrow{\$} \mathrm{Func}(X, Y)$ where $\mathrm{Func}(X, Y)$ denotes the set of all functions from X to Y), by computationally-bounded adversaries.

To be more precise, let A denote an adversary trying to distinguish between f and φ. That is, A is given access to either the "real" oracle f or the "random" oracle φ. The f-oracle picks a random key $k \xleftarrow{\$} K$ at the beginning of each experiment and, upon a query $x \in X$ made by A, returns the value $y = f_k(x)$ to A. On the other hand, the φ-oracle picks a random function $\varphi \xleftarrow{\$} \mathrm{Func}(X, Y)$ at the beginning of each experiment and returns the value $y = \varphi(x)$ upon a query $x \in X$. Then the *advantage* of adversary A is defined by

$$\mathrm{Adv}_f^{\mathrm{prf}}(A) \overset{\mathrm{def}}{=} \Pr\big[A^f \Rightarrow 1\big] - \Pr\big[A^\varphi \Rightarrow 1\big],$$

where by the notation $A^{\mathcal{O}} \Rightarrow 1$ we denote the event that A, given access to oracle $\mathcal{O}$, outputs value 1.

In order for the advantage function to be well-defined, the resources of adversary A need to be bounded. We define

$$\mathrm{Adv}_f^{\mathrm{prf}}(t, q, \ell) \overset{\mathrm{def}}{=} \max_A \mathrm{Adv}_f^{\mathrm{prf}}(A),$$

where max runs over all adversaries, whose time complexity is at most t, making at most q oracle queries, each query being at most length ℓ (in some appropriate units). In order to measure the time complexity t, we fix some model of computation. The time complexity includes the maximum time for adversary A to execute each overlying experiment, including the time consumed by oracles, plus the code size of A. If f accepts only fixed-length inputs, then the quantity ℓ is simply omitted from the notation.

Quasi-random Functions (QRFs). The notion of QRFs is an information-theoretic version of that of PRFs [16]. A QRF $\boldsymbol{\psi}$ is a family of functions, indexed not by a key but by smaller random function(s). An adversary A attacking ψ is computationally unbounded. The advantage function is defined similarly:

$$\mathrm{Adv}_{\psi}^{\mathbf{qrf}}(A) \overset{\text{def}}{=} \Pr\big[A^{\psi} \Rightarrow 1\big] - \Pr\big[A^{\varphi} \Rightarrow 1\big],$$

and we also define

$$\mathrm{Adv}_{\psi}^{\mathbf{qrf}}(q, \ell) \overset{\text{def}}{=} \max_{A} \mathrm{Adv}_{\psi}^{\mathbf{qrf}}(A),$$

where again, ℓ may be omitted from the notation if irrelevant.

Tweaking Pseudo-random Functions. Here we recast the theory of tweakable block ciphers in the context of compression functions. In fact, developing the theory in the style of compression function is easier, because block ciphers are permutations, whilst compression functions are functions, which in particular means that we do not need to exercise the PRP ↔ PRF Switching Lemma. Also, we utilize the condition $b \geq 2n$ here, which is something impossible with block ciphers where there exists an innate relation $b = n$.

The purpose of tweaking a PRF f_k is to construct many functions $f_1, f_2, \ldots$ from f which are indistinguishable from a collection of (truly) random functions $\varphi_1, \varphi_2, \ldots$. In order to do this, we begin with defining an initial value Δ_0 of masks to be the leftmost b bits of

$$f_k(1) \parallel f_k(2) \parallel \cdots \parallel f_k(\lceil b/n \rceil),$$

where integers $1, 2, \ldots, \lceil b/n \rceil$ are represented as b-bit strings by some canonical encoding. We then modify this value Δ_0 sequentially, by "incrementing" as

$$\Delta_1\,,\ \Delta_2\,,\ \ldots\,,\ \Delta_\ell,$$

up until about $\ell \approx 2^n$. It is essential here that the values $\Delta_1, \Delta_2, \ldots, \Delta_\ell$ are all distinct. In addition, we also need a "special" set of offsets

$$\bar{\Delta}_{L,1}\,,\ \bar{\Delta}_{L,2}\,,\ \bar{\Delta}_{L,3},$$

for each $L \in \{1, 2, \ldots, \ell\}$. All of these values need to be distinct among themselves and from the above list of ℓ-many values.

In our construction a message $M \in \{0,1\}^*$ needs to be padded so that the length becomes a multiple of b bits, before being processed. This would cause the length to increase by b bits in case $|M|$ is already a multiple of b. If one wants to avoid the extra block of computation when $|M|$ happens to be exactly equal to a multiple of b bits, then one needs another special set of offsets

$$\bar{\bar{\Delta}}_{L,1}\,,\ \bar{\bar{\Delta}}_{L,2}\,,\ \bar{\bar{\Delta}}_{L,3},$$

for performance optimization (saving one block of computation). For the sake of simplicity, we do not make use of such masks $\bar{\bar{\Delta}}_1, \bar{\bar{\Delta}}_2, \bar{\bar{\Delta}}_3$ and do contend ourselves with the trivial padding $M \| 10^*$. Our construction always requires three blocks of extra computation in any event, so the effectiveness of such optimization is limited. All the proofs carry over with such optimization but only become more complicated.

Incrementing Masks. It remains to describe the ways of "incrementing" the masks. There are several known methods [17,18,19,20], and some of them can be transformed into the context of compression functions. In the following we modify the method in [20] so that it becomes compatible with our construction.

The basic framework of [20] is to let $\varDelta_i \overset{\mathrm{def}}{=} \alpha^i \cdot \varDelta_0$, where the multiplication is done in the finite field $\mathbb{F}_{2^b}$, and $\alpha \in \mathbb{F}_{2^b}^{\times}$ is a non-zero element whose multiplicative order is large enough (say $\geq 2^n$). The functions f_i are created via $f_i(m) \overset{\mathrm{def}}{=} f_k(m \oplus \varDelta_i)$. The special offsets are created via $\bar{\varDelta}_{L,j} \overset{\mathrm{def}}{=} \alpha^L \cdot \beta^j \cdot \varDelta_0$, where $\beta \in \mathbb{F}_{2^b}^{\times}$ is an element such that $\alpha^L \beta^j$ can be guaranteed to be distinct from α^is. A preferred choice of α, β is usually $\alpha = 2$ and $\beta = 3$.

The finite field needs to be represented by an irreducible polynomial $g(x) \in \mathbb{F}_2[x]$ of degree b, with $\alpha = x(=$ "2") being a generator of $\mathbb{F}_{2^b}^{\times}$ (so that its multiplicative order is $2^b - 1$). Then we compute $\log_x(x+1)$ in this field and verify that it is huge, which enables us to choose $\beta = x + 1(=$ "3"). Computing such discrete logarithms for block ciphers has been feasible owing to small parameters such as $b = 64$ and $b = 128$ [20].

Yet, now we are dealing with a compression function with a parameter such as $b = 512$, which most likely stops us from computing such discrete logarithms. So instead we choose an irreducible polynomial $g(x)$ so that $\alpha = 2$ generates only a subgroup of $\mathbb{F}_{2^b}^{\times}$ but its order being large enough ($\geq 2^n$). Then we merely need to verify that $\beta = 3$ generates the subgroup "missed" by $\alpha = 2$.

For example, consider the case $b = 512$ and $n = 128$. We are then working in the multiplicative group $\mathbb{F}_{2^{512}}^{\times}$ of the field with 2^{512} elements, and the order of the group $2^{512} - 1$ can be factored as $2^{512} - 1 = (2^1+1)(2^2+1)\cdots(2^{128}+1)(2^{256}+1)$. In particular, the term $2^{128} + 1$ can be further factored as [21]:

$$2^{128} + 1 = 59649589127497217 \times 5704689200685129054721.$$

It can be directly verified that these two prime factors appear nowhere else in the factorization of $2^{512} - 1$.

Now we choose $x^{512} + x^{12} + x^7 + x^2 + 1 \in \mathbb{F}_2[x]$ as an irreducible polynomial to represent the field $\mathbb{F}_{2^{512}}$ and verify that $x^{(2^{512}-1)/59649589127497217} \neq 1$ and $x^{(2^{512}-1)/5704689200685129054721} \neq 1$ in this field, which ensures that the multiplicative order of the element x is at least $2^{128} + 1$. On the contrary, notice that $x^{(2^{512}-1)/17} = 1$, where $17 = 2^4 + 1$ appears only once in the factorization of $2^{512} - 1$, from which we deduce that the element x does not "generate" the subgroup of order 17 in the multiplicative group $\mathbb{F}_{2^{512}}^{\times}$. On the other hand, observe that $(x+1)^{(2^{512}-1)/17} \neq 1$, which implies that the group generated by $x+1$ <u>does</u> contain the subgroup of order 17.

After the above verification we are able to set

$$\varDelta_i \overset{\mathrm{def}}{=} x^i \varDelta_0 \text{ and } \bar{\varDelta}_{L,j} \overset{\mathrm{def}}{=} x^L (x+1)^j \varDelta_0$$

for $i, L \in \{1, 2, \ldots, 2^{128}\}$ and $j \in \{1, 2, 3\}$. These masks are all distinct because of the following three reasons: (1) We have $\varDelta_i \neq \varDelta_{i'}$ if $i \neq i'$, owing to the high order of the element x; (2) We have $\varDelta_i \neq \bar{\varDelta}_{L,j}$ for any i, L, j in the above ranges, because $x^L(x+1)^j$ generates a group that contains the subgroup of order 17 while x^i does not; (3) We have $\bar{\Delta}_{L,j} \neq \bar{\Delta}_{L',j'}$ as long as $(L, j) \neq (L', j')$, for if the equality $x^L(x+1)^j\Delta_0 = x^{L'}(x+1)^{j'}\Delta_0$ holds in the field with i, L, j being in the above ranges, then by looking at the subgroup of order 17 we see that $j = j'$, which immediately implies that $L = L'$.

Lemma 1. *If f is a PRF and the masks $\Delta_1, \Delta_2, \ldots, \Delta_\ell \in \{0,1\}^b$ are all distinct, created via $\Delta_i \overset{\text{def}}{=} \gamma_i \cdot \Delta_0 \in \mathbb{F}_{2^b}$ with γ_i being some (public) function of i independent of the value Δ_0, then the functions $f_1, f_2, \ldots, f_\ell$ defined by $f_i(m) \overset{\text{def}}{=} f_k(m \oplus \Delta_i)$ are indistinguishable from random functions $\varphi_1, \varphi_2, \ldots, \varphi_\ell$, by an adversary having time complexity at most t and making at most $q \geq \lceil b/n \rceil$ queries to each f_i (or φ_i), except for the probability at most*

$$\mathrm{Adv}_f^{\mathrm{prf}}(t, q') + \frac{q^2}{2^{2n-1}},$$

where $q' = (\ell + 1)q$.

Proof. The proof is done via hybrid argument. Consider an intermediate oracle $\boldsymbol{\rho}$ which chooses a random function $\rho : \{0,1\}^b \to \{0,1\}^n$ at the beginning of each experiment and upon a query m to f_i returns $\boldsymbol{\rho_i}(m) \overset{\text{def}}{=} \rho(m \oplus \Delta_i)$ instead. Here, Δ_0 is computed as the leftmost b bits of

$$\rho(1) \parallel \rho(2) \parallel \cdots \parallel \rho(\lceil b/n \rceil),$$

and the masks $\Delta_1, \Delta_2, \ldots, \Delta_\ell$ are generated accordingly, which are all distinct as long as $\Delta_0 \neq 0^b$.

Now let A be an adversary trying to distinguish between $f_1, f_2, \ldots, f_\ell$ and $\varphi_1, \varphi_2, \ldots, \varphi_\ell$. Assume that A has time complexity at most t and makes at most q queries to each f_i (or φ_i). It is straightforward to see that the probability that A distinguish between $f_1, f_2, \ldots, f_\ell$ and $\boldsymbol{\rho_1}, \boldsymbol{\rho_2}, \ldots, \boldsymbol{\rho_\ell}$ is at most

$$\mathrm{Adv}_f^{\mathrm{prf}}(t, q'),$$

where $q' \overset{\text{def}}{=} (\ell + 1)q \geq \ell q + \lceil b/n \rceil$.

We next show that $\boldsymbol{\rho}$ is quasi-random. Observe that functions $\boldsymbol{\rho_1}, \boldsymbol{\rho_2}, \ldots, \boldsymbol{\rho_\ell}$ behave just like random functions $\varphi_1, \varphi_2, \ldots, \varphi_\ell$ unless one of the following events occurs: (1) $\Delta_0 = 0^b$, or (2) A "collision" occurs among the inputs to ρ and $\boldsymbol{\rho_i}$. The probability for event (1) to occur is exactly $2^{-b} \leq 2^{-2n}$. For (2), if a collision occurs between inputs to ρ and $\boldsymbol{\rho_i}$, then it means that $j = m \oplus \Delta_i = m \oplus \gamma_i \Delta_0$ for some $j \in \{1, 2, \ldots, \lceil b/n \rceil\}$. This yields $(j \oplus m)/\gamma_i = \Delta_0$, and for a fixed (j, i) the probability of such an event is $2^{-b} \leq 2^{-2n}$. On the other hand, if a collision occurs between an input to $\boldsymbol{\rho_i}$ and an input to $\boldsymbol{\rho_j}$ for some $1 \leq i < j \leq \ell$, then it means that we have $m \oplus \Delta_i = m' \oplus \Delta_j$, or equivalently $m \oplus \gamma_i \Delta_0 = m' \oplus \gamma_j \Delta_0$. This yields $(m \oplus m')/(\gamma_i \oplus \gamma_j) = \Delta_0$, and for a fixed (i, j) the probability that such an event occurs is $2^{-b} \leq 2^{-2n}$.

Since the values returned by $\boldsymbol{\rho}$ are random, adversary A learns nothing from them to bring about a collision. That is, we can assume that A is non-adaptive and outputs a sequence of fixed values $(i_1, m_1), (i_2, m_2), \ldots, (i_q, m_q)$, hoping that a collision occurs among them [16]. Now for the first type of collision there are at most $\lceil b/n \rceil \cdot q$ possible pairs, while for the second type there are at most $\binom{q}{2}$ pairs. Thus the advantage that A distinguish between $\boldsymbol{\rho_1}, \boldsymbol{\rho_2}, \ldots, \boldsymbol{\rho_\ell}$ and $\varphi_1, \varphi_2, \ldots, \varphi_\ell$ is at most

$$\frac{1}{2^{2n}} + \lceil b/n \rceil \cdot q \cdot \frac{1}{2^{2n}} + \binom{q}{2} \cdot \frac{1}{2^{2n}} \leq \frac{q^2}{2^{2n-1}}.$$

□

4 Description of the Proposed Mode

In this section we give the definition of our algorithm. Recall that our starting primitive is a compression function $f : \{0,1\}^{n+b} \to \{0,1\}^n$. We key it via $f_k(m) \stackrel{\text{def}}{=} f(k\|m)$ and tweak it via $f_i(m) \stackrel{\text{def}}{=} f_k(m \oplus \Delta_i)$, obtaining

$$f_1\,,\, f_2\,,\, \ldots\,,\, f_\ell\,,\, \bar{f}_1\,,\, \bar{f}_2\,,\, \bar{f}_3,$$

which should be (computationally) indistinguishable from random functions $\varphi_1, \varphi_2, \ldots, \varphi_\ell, \bar{\varphi}_1, \bar{\varphi}_2, \bar{\varphi}_3$ (Recall that f_i depends on the choice of key k, while $\bar{f}_i$ depends on the message length L, and so does $\bar{\varphi}_i$).

```
Algorithm $\bar{f}_{123}(S\|v_L\|s)$    // $S \in \{0,1\}^b$, $v_L, s \in \{0,1\}^n$
    Set $u \leftarrow v_L\|s$
    Compute $\Sigma_1 \leftarrow \bar{f}_1(S)$ and $\Sigma_2 \leftarrow \bar{f}_2(S)$
    Set $w \leftarrow (\Sigma_1\|\Sigma_2) \oplus u$
    Output $\tau \leftarrow \bar{f}_3(w\|0^{b-2n})$

Algorithm $F_k(M)$    // $M \in \{0,1\}^*$
    Pad $M \leftarrow M\|10^*$
    Divide $M = m_1\|m_2\| \cdots \|m_L$ so that $m_i \in \{0,1\}^b$
    Compute checksum $S \leftarrow \bigoplus_{i=1}^{L} m_i$
    Initialize $v_0 \leftarrow 0^n$
    Iterate $v_i \leftarrow f_i\big(m_i \oplus (v_{i-1}\|0^{b-n})\big)$ for $i = 1, 2, \ldots, L$
    Compute checksum $s \leftarrow \bigoplus_{i=1}^{L} v_i$
    Output $\tau \leftarrow \bar{f}_{123}(S\|v_L\|s)$
```

Fig. 1. Definitions of $\bar{f}_{123}$ and F_k

Now with these tweaked functions in hand, we first define a function (which depends on the choice of L)

$$\bar{f}_{123} : \{0,1\}^{b+2n} \to \{0,1\}^n,$$

from the three functions $\bar{f}_1$, $\bar{f}_2$ and $\bar{f}_3$. This function is used at the end of processing a message in our mode of operation

$$F_k : \{0,1\}^* \to \{0,1\}^n.$$

See Fig. 1 for precise definitions, as well as Fig. 2 for a pictorial description.

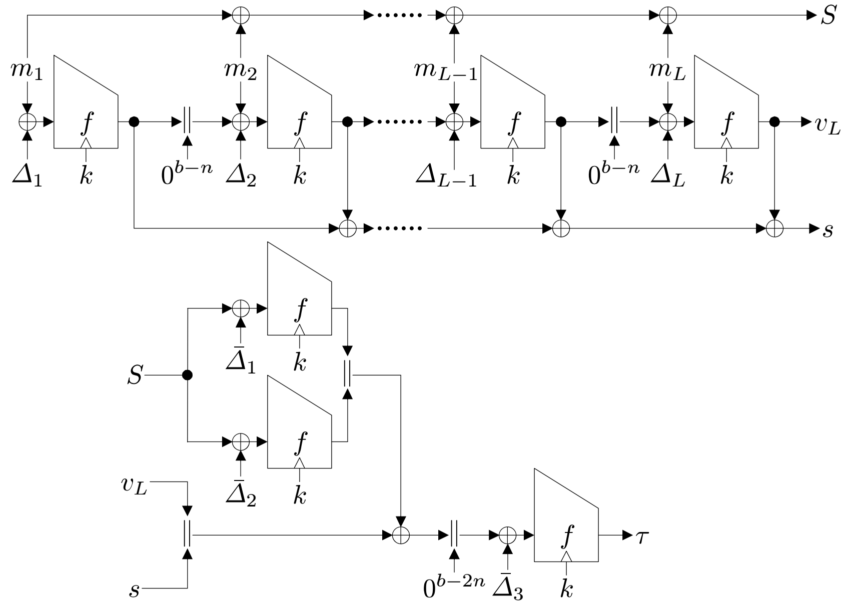


Fig. 2. Proposed mode of operation F_k (the lower half corresponding to $\bar{f}_{123}$)

The construction of $\bar{f}_{123}$ may look unnatural at first glance. We note that this is not the only one that works. For example, the roles of S and $v_L \| s$ may be switched, or Two-Lane construction [9] may be used in the place. Our choice of $\bar{f}_{123}$ simply comes from considerations of efficiency.

The major feature of our mode of operation is the usage of message checksum $S = \bigoplus_{i=1}^{L} m_i$ and intermediate-value checksum $s = \bigoplus_{i=1}^{L} v_i$. The checksum construction is effectively combined with the tweaked compression functions, yielding security beyond the birthday barrier.

5 Proofs of Security beyond the Birthday Barrier

We want to prove that our mode of operation F_k is (computationally) indistinguishable from a truly random function $\varPsi : \{0,1\}^* \to \{0,1\}^n$ in such a way as its security is still guaranteed when $q \approx 2^{n/2}$. Succinctly, we prove the following theorem:

Theorem 1. *Let $F_k : \{0,1\}^* \to \{0,1\}^n$ be the mode of operation as defined in Sect. 4. It is a PRF without the birthday barrier if the underlying compression function is a PRF. Concretely, we have*

$$\mathrm{Adv}_F^{\mathrm{prf}}(t, q, \ell) \le \mathrm{Adv}_f^{\mathrm{prf}}(t, q') + \frac{(\ell+5)q^2}{2^{2n+1}},$$

where $q \ge \lceil b/n \rceil$ and $q' = (\ell+4)q$.

The proof is based on hybrid argument. In order to prove that F_k is a PRF via hybrid argument, we construct intermediate QRFs $\boldsymbol{\Phi}$ and $\boldsymbol{\Phi}_{\boldsymbol{\psi}}$ as below.

The QRF $\boldsymbol{\Phi} : \{0,1\}^* \to \{0,1\}^n$ is constructed as follows. In the definition of F_k, we replace functions $f_1, f_2, \ldots, f_\ell$ with random functions $\varphi_1, \varphi_2, \ldots, \varphi_\ell$, where $\varphi_i : \{0,1\}^b \to \{0,1\}^n$ is drawn independently at random. We also replace $\bar{f}_1, \bar{f}_2, \bar{f}_3$ with random functions $\bar{\varphi}_1, \bar{\varphi}_2, \bar{\varphi}_3$ (The choice of these random functions depends on the value L). This gives us a (to-be-proven) QRF $\boldsymbol{\Phi}$. See Fig. 3 for an illustration of $\boldsymbol{\Phi}$.

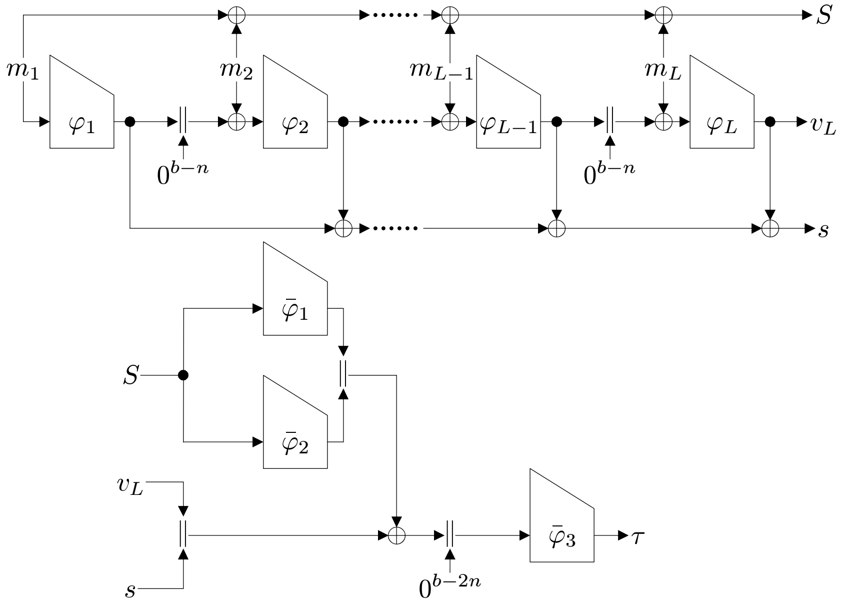


Fig. 3. Description of $\boldsymbol{\Phi}$ (the lower half corresponding to $\bar{\boldsymbol{\varphi}}_{\mathbf{123}}$)

The other QRF $\boldsymbol{\Phi}_{\boldsymbol{\psi}}$ is obtained by modifying the last component in $\boldsymbol{\Phi}$. In the definition of $\boldsymbol{\Phi}$, note that we have a (to-be-proven) QRF

$$\bar{\boldsymbol{\varphi}}_{\mathbf{123}} : \{0,1\}^{b+2n} \to \{0,1\}^n,$$

which is constructed of $\bar{\varphi}_1, \bar{\varphi}_2, \bar{\varphi}_3$ (Needless to say, $\bar{\boldsymbol{\varphi}}_{\mathbf{123}}$ in $\boldsymbol{\Phi}$ corresponds to $\bar{f}_{123}$ in F_k). We replace this QRF $\bar{\boldsymbol{\varphi}}_{\mathbf{123}}$ with a truly random function

$$\psi : \{0,1\}^{b+2n} \to \{0,1\}^n.$$

That is to say, for each value of L, the function $\bar{\boldsymbol{\varphi}}_{\mathbf{123}}$ is replaced with a new random function $\psi = \psi_L$. We name the resulting scheme as $\boldsymbol{\Phi}_{\boldsymbol{\psi}} : \{0,1\}^* \to \{0,1\}^n$.

Now the hybrid argument works as follows. Let A be an adversary trying to distinguish between F_k and Ψ. Then

$$\begin{aligned}\mathrm{Adv}_F^{\mathrm{prf}}(A) &\overset{\mathrm{def}}{=} \Pr\bigl[A^F \Rightarrow 1\bigr] - \Pr\bigl[A^{\Psi} \Rightarrow 1\bigr] \\ &= \Pr\bigl[A^F \Rightarrow 1\bigr] - \Pr\bigl[A^{\boldsymbol{\Phi}} \Rightarrow 1\bigr] \\ &\quad + \Pr\bigl[A^{\boldsymbol{\Phi}} \Rightarrow 1\bigr] - \Pr\bigl[A^{\boldsymbol{\Phi}_\psi} \Rightarrow 1\bigr] \\ &\quad\quad + \Pr\bigl[A^{\boldsymbol{\Phi}_\psi} \Rightarrow 1\bigr] - \Pr\bigl[A^{\Psi} \Rightarrow 1\bigr].\end{aligned}$$

We bound the three differences in the rest of this section.

To evaluate the first difference, we note that it is rather straightforward to see that

$$\Pr\bigl[A^F \Rightarrow 1\bigr] - \Pr\bigl[A^{\boldsymbol{\Phi}} \Rightarrow 1\bigr] \le \mathrm{Adv}_f^{\mathrm{prf}}(t, q') + \frac{q^2}{2^{2n-1}},$$

where $q' \overset{\mathrm{def}}{=} (\ell + 4)q$. This is because distinguishing between F and Φ essentially amounts to the security of tweaked functions $f_1, f_2, \ldots, f_\ell, \bar{f}_1, \bar{f}_2, \bar{f}_3$, where each $\bar{f}_i$ may vary upon each query (of varying length). So the above inequality follows from Lemma 1.

We next bound the second difference. It is again easy to see that

$$\Pr\bigl[A^{\boldsymbol{\Phi}} \Rightarrow 1\bigr] - \Pr\bigl[A^{\boldsymbol{\Phi}_\psi} \Rightarrow 1\bigr] \le \mathrm{Adv}_{\bar{\boldsymbol{\varphi}}_{\mathbf{123}}}^{\mathbf{qrf}}(q).$$

This is because any adversary trying to distinguish between $\boldsymbol{\Phi}$ and $\boldsymbol{\Phi}_{\boldsymbol{\psi}}$ essentially amounts to distinguishing between $\bar{\boldsymbol{\varphi}}_{\mathbf{123}}$ and ψ. So it remains to evaluate the quantity $\mathrm{Adv}_{\bar{\boldsymbol{\varphi}}_{\mathbf{123}}}^{\mathbf{qrf}}(q)$. We do this in the following lemma:

Lemma 2. *Fix L. Then the function $\bar{\boldsymbol{\varphi}}_{\mathbf{123}}$ is a quasi-random function. More concretely, we have*

$$\mathrm{Adv}_{\bar{\boldsymbol{\varphi}}_{\mathbf{123}}}^{\mathbf{qrf}}(q) \le \frac{q^2}{2^{2n+1}}.$$

Proof. Let B be an adversary trying to distinguish between $\bar{\boldsymbol{\varphi}}_{\mathbf{123}}$ and a truly random function $\psi : \{0,1\}^{b+2n} \to \{0,1\}^n$. Since $\bar{\varphi}_3$ is a random function, $\bar{\boldsymbol{\varphi}}_{\mathbf{123}}$ behaves just like a truly random function unless a collision occurs among the inputs to $\bar{\varphi}_3$. By a "collision" we mean an event $w = w'$ for distinct inputs $S\|u \ne S'\|u'$ (We carry over the symbols such as w, S, u from the definition of $\bar{f}_{123}$ in Fig. 1). We want to evaluate the probability that such an event occurs.

Since the values returned by $\bar{\boldsymbol{\varphi}}_{\mathbf{123}}$ are random, and B learns nothing from the values in order to bring about a collision, without loss of generality we can assume that B is non-adaptive [16]. That is to say, B just queries a sequence of fixed values $S_1\|u_1, S_2\|u_2, \ldots, S_q\|u_q$, hoping that a "collision" occurs between w_i and w_j for some $1 \le i < j \le q$.

So suppose $S\|u \ne S'\|u'$ and $w = w'$. We claim that the probability that such an event occurs is at most 2^{-2n}. To see this, we first observe that $S \ne S'$, for if $S = S'$, then $\Sigma_1\|\Sigma_2 = \Sigma_1'\|\Sigma_2'$ and $u \ne u'$, which implies $w \ne w'$ and hence never a collision. Thus we are looking at an event such that $\Sigma_1 \oplus v_L = \Sigma_1' \oplus v_L'$ <u>and</u> $\Sigma_2 \oplus s = \Sigma_2' \oplus s'$ for some fixed S, S', v_L, v_L', s, s'. Since $\bar{\varphi}_1$ and $\bar{\varphi}_2$ are random functions, the probability that each event occurs is 2^{-n}. Moreover, since $\bar{\varphi}_1$ and $\bar{\varphi}_2$ are independently random, the probability that both events occur is $2^{-n} \cdot 2^{-n} = 2^{-2n}$.

We have seen that the probability that $S_i \| u_i \neq S_j \| u_j$ and $w_i = w_j$ is at most 2^{-2n}. Since there are at most $\binom{q}{2}$ choices of values (i, j), we conclude that

$$\mathrm{Adv}^{\mathbf{qrf}}_{\boldsymbol{\varphi}_{\mathbf{123}}}(q) \leq \binom{q}{2} \cdot \frac{1}{2^{2n}} \leq \frac{q^2}{2^{2n+1}}.$$

□

Now note that A's varying lengths L of its queries does not contribute to increasing the collision probability. So we obtain

$$\Pr\bigl[A^{\boldsymbol{\Phi}} \Rightarrow 1\bigr] - \Pr\bigl[A^{\boldsymbol{\Phi}_{\psi}} \Rightarrow 1\bigr] \leq \frac{q^2}{2^{2n+1}}.$$

Lastly, we bound the third difference. This is nothing but the quantity

$$\mathrm{Adv}^{\mathbf{qrf}}_{\boldsymbol{\Phi}_{\psi}}(A) \stackrel{\mathrm{def}}{=} \Pr\bigl[A^{\boldsymbol{\Phi}_{\psi}} \Rightarrow 1\bigr] - \Pr\bigl[A^{\Psi} \Rightarrow 1\bigr],$$

by definition. Hence in the next lemma we show that $\boldsymbol{\Phi}_{\psi}$ is indeed quasi-random:

Lemma 3. *The function $\boldsymbol{\Phi}_{\psi}$ is quasi-random. More concretely, we have*

$$\mathrm{Adv}^{\mathbf{qrf}}_{\boldsymbol{\Phi}_{\psi}}(q, \ell) \leq \frac{\ell q^2}{2^{2n+1}}.$$

Proof. Since $\psi : \{0,1\}^{b+2n} \to \{0,1\}^n$ is a random function, $\boldsymbol{\Phi}_{\psi}$ behaves just like a truly random function except when a collision occurs among the inputs to ψ. Here by a "collision" we mean an event that for two distinct queries $M = m_1 \| m_2 \| \cdots \| m_L$ and $M' = m_1' \| m_2' \| \cdots \| m_{L'}'$ the equality $S \| v_L \| s = S' \| v_{L'}' \| s'$ holds.

We want to evaluate the probability that such a collision occurs. We divide our proof into two cases, depending on the lengths L, L' of two messages.

Case A: $\boldsymbol{L \neq L'}$. There is nothing to prove in this case. That is, since the choice of ψ changes for different values of L, two independently random functions, say ψ_L and $\psi_{L'}$, are used for messages of different lengths. So there is no "collision" to consider here; the two outputs are truly random.

Case B: $\boldsymbol{L = L'}$. Observe that from the condition $M \neq M'$ there exists a unique $a \in \{1, 2, \ldots, L\}$ such that $(v_{a-1}, m_a) \neq (v_{a-1}', m_a')$ and $(v_{i-1}, m_i) = (v_{i-1}', m_i')$ holds for $i = a+1, a+2, \ldots, L$.

Case B-1: $\boldsymbol{(v_{a-1} \| 0^{b-n}) \oplus m_a = (v_{a-1}' \| 0^{b-n}) \oplus m_a'}$. In this case we note that the rightmost $b - n$ bits of m_a and m_a' must be identical, and with the condition $(v_{a-1}, m_a) \neq (v_{a-1}', m_a')$ we see that $v_{a-1} \neq v_{a-1}'$ <u>and</u> $m_a \neq$

m'_a. Since $v_{a-1} \neq v'_{a-1}$, the two inputs to the random function φ_{a-1} must differ, implying that v_{a-1} and v'_{a-1} are two independently random values. It means that the equality $(v_{a-1} \| 0^{b-n}) \oplus m_a = (v'_{a-1} \| 0^{b-n}) \oplus m'_a$ holds with a probability of 2^{-n}. Moreover, observe that since $s = s'$ and $v_i = v'_i$ for $a \leq i \leq L$ we must have $\bigoplus_{i=1}^{a-1} v_i = \bigoplus_{i=1}^{a-1} v'_i$. The condition $v_{a-1} \neq v'_{a-1}$ also tells us that $\bigoplus_{i=1}^{a-2} v_i \neq \bigoplus_{i=1}^{a-2} v'_i$. Now put $s_{a-2} \overset{\text{def}}{=} \bigoplus_{i=1}^{a-2} v_i$ and $s'_{a-2} \overset{\text{def}}{=} \bigoplus_{i=1}^{a-2} v'_i$. Then the values s_{a-2} and s'_{a-2} are created using random functions $\varphi_1, \varphi_2, \ldots, \varphi_{a-2}$, which are all independent from the random function φ_{a-1}. Therefore, the equality $s_{a-2} \oplus v_{a-1} = s'_{a-2} \oplus v'_{a-1}$ holds with a probability of 2^{-n}. This event is clearly independent from the previous equality, so this case occurs with a probability at most $2^{-n} \cdot 2^{-n} = 2^{-2n}$.

Case B-2: $(v_{a-1} \| 0^{b-n}) \oplus m_a \neq (v'_{a-1} \| 0^{b-n}) \oplus m'_a$. In this case the inputs to the random function φ_a are different, but their outputs are colliding (*i.e.*, $v_a = v'_a$). Clearly, the probability that such an event occurs is exactly 2^{-n}.

Case B-2-(i): $v_{a-1} \neq v'_{a-1}$. In this case we do an analysis similar to Case B-1. The condition $v_{a-1} \neq v'_{a-1}$ tells us that $\bigoplus_{i=1}^{a-2} v_i \neq \bigoplus_{i=1}^{a-2} v'_i$. Put $s_{a-2} \overset{\text{def}}{=} \bigoplus_{i=1}^{a-2} v_i$ and $s'_{a-2} \overset{\text{def}}{=} \bigoplus_{i=1}^{a-2} v'_i$. Then we have $s_{a-2} \neq s'_{a-2}$, and the equality $s_{a-2} \oplus v_{a-1} = s'_{a-2} \oplus v'_{a-1}$ holds with a probability of 2^{-n}.

Case B-2-(ii): $v_{a-1} = v'_{a-1}$. In this case we have $s_{a-1} = s'_{a-1}$. Since $M \neq M'$ and $S = \bigoplus_{i=1}^{L} m_i = S' = \bigoplus_{i=1}^{L'} m'_i$, there must exist at least two values of $i \in \{1, 2, \ldots, L\}$ such that $m_i \neq m'_i$. One of such values may be equal to the value a, but it still guarantees that there exists a $b \in \{1, 2, \ldots, a-1\}$ such that $(v_{b-1}, m_b) \neq (v'_{b-1}, m'_b)$ and $v_i = v'_i$, $s_i = s'_i$ for $i = b, b+1, \ldots, a-1$ and $m_i = m'_i$ for $i = b+1, b+2, \ldots, a-1$. Then we do an analysis at block b similar to Case B-1 and B-2 as done at block a, in order to prove that such an event happens at block b with a probability at most 2^{-n}.

In all events we see that the collision probability in Case B-2 is at most $2^{-n} \cdot 2^{-n} = 2^{-2n}$.

We have shown that in all cases the collision probability is at most 2^{-2n}. Since the values returned by ψ are random, and A learns nothing from these values in bringing about a collision, we can assume that A is non-adaptive. So assume that A makes a fixed sequence of queries $M_1, M_2, \ldots, M_q$, hoping that a collision occurs at some $1 \leq i < j \leq q$. We have just seen that for a pair (M_i, M_j) the probability that the two messages collide is at most 2^{-2n}. Since there are at most $\binom{q}{2}$ pairs, we conclude that

$$\mathrm{Adv}^{\mathbf{qrf}}_{\boldsymbol{\Phi}_\psi}(q, \ell) \leq \ell \cdot \binom{q}{2} \cdot \frac{1}{2^{2n}} < \frac{\ell q^2}{2^{2n+1}}.$$

□

Now we go back to proving our main theorem. We have

$$\begin{aligned}\mathrm{Adv}_F^{\mathrm{prf}}(A) &\overset{\mathrm{def}}{=} \Pr\big[A^F \Rightarrow 1\big] - \Pr\big[A^{\Psi} \Rightarrow 1\big]\\ &= \Pr\big[A^F \Rightarrow 1\big] - \Pr\big[A^{\boldsymbol{\Phi}} \Rightarrow 1\big]\\ &\quad + \Pr\big[A^{\boldsymbol{\Phi}} \Rightarrow 1\big] - \Pr\big[A^{\boldsymbol{\Phi}_{\psi}} \Rightarrow 1\big]\\ &\qquad + \Pr\big[A^{\boldsymbol{\Phi}_{\psi}} \Rightarrow 1\big] - \Pr\big[A^{\Psi} \Rightarrow 1\big]\\ &\le \mathrm{Adv}_f^{\mathrm{prf}}(t, q') + \frac{q^2}{2^{2n-1}} + \frac{q^2}{2^{2n+1}} + \frac{\ell q^2}{2^{2n+1}}\\ &= \mathrm{Adv}_f^{\mathrm{prf}}(t, q') + \frac{(\ell+5)q^2}{2^{2n+1}},\end{aligned}$$

where $q' = (\ell + 4)q$. This proves our main theorem.

6 Optimization for Better Performance

In this section we introduce a couple of techniques to improve the performance of our mode. One is associated with the methods of setting up the masks, and the other is related to the ways of keying the compression function.

Mask Partition. The performance of our mode should be essentially as good as that of a naively-chained construction such as the Merkle-Damgård iteration (HMAC), except for the computational costs of workings outside the underlying primitive f. These include concatenation, XOR, mask setup (initialization) and its incrementation. The last calculation can be realized with a 1-bit (left-)shift operation plus a conditional XOR, because an incrementation corresponds to multiplying x to the mask in the field $\mathbb{F}_{2^b}$ (multiplication by $x + 1$ requires slightly more operations).

The 1-bit shift operation may be costly in software implementations, because we need to perform the operation on a long mask, say $b = 512$ bits, while the available size of registers may be much smaller, say 32 bits. The long-size mask causes another problem that we may be forced to store data outside registers, further lowering performance. These difficulties can be relaxed by dividing the b-bit mask into copies of a $2n$-bit mask. For example, consider the case $b = 512$ and $n = 128$. Then we can use $\varDelta_i \| \varDelta_i$ as the mask, where $\varDelta_i$ is a 256-bit mask (using for example $x^{256} + x^{16} + x^3 + x^2 + 1 \in \mathbb{F}_2[x]$ as an irreducible polynomial). Note that our proofs work with such a construction without significant changes.

Key-Length Flexibility. Our mode does not require re-keying, presenting a contrast to the classical Merkle-Damgård iteration that re-keys at every step. This does not have an impact on performance with compression functions such as sha1 and sha256, but the situation would be quite different with block-cipher-like primitives equipped with heavy key-schedule algorithms.

We remark that our construction has no restriction on the key space, though so far we have assumed $k \in \{0,1\}^n$. In fact, our construction works with any finite PRF $f_k : \{0,1\}^b \to \{0,1\}^n$ with $k \in K$, where K can be an arbitrary type of key space, as long as f is a secure PRF. Hence using a key k shorter than n bits speeds up performance (*i.e.*, each invocation to f processes more bits of a message). This sort of situation may occur when the desired key length does not match the value n of a compression function in hand.

7 Open Problems

Case $b < 2n$ and Block-Cipher-Based Construction. Our construction requires that the underlying compression function $f : \{0,1\}^{n+b} \to \{0,1\}^n$ should meet the condition $b \geq 2n$. We leave it as an open problem whether we can construct a mode of operation, meeting our goals, with a compression function f with $b < 2n$. Since we utilize this condition essentially in two different places, our method does not seem to be feasible with such compression functions. In particular, the last process with $\bar{f}_{123}$ may be constructed by methods such as [12,13,14], but using the condition in tweaking f seems to face a hard problem.

A possibly more challenging problem is to construct a mode of operation using an f_k with $b = n$ and each f_k being a permutation, rather than a function (*i.e.*, a block cipher). This introduces the difficulty in handling the PRP ↔ PRF Switching Lemma that causes the birthday security degradation.

Parallelizable Construction. Our construction is inherently serial, and thus not parallelizable. Parallelizability is one of the desirable properties in constructing a mode of operation.

Recall that PMAC [18] is a mode of operation for message authentication, which is fully parallelizable. Although usually constructed of a block cipher, PMAC can be based on a compression function $f : \{0,1\}^{n+b} \to \{0,1\}^n$ meeting the condition $b \geq 2n$. We can then modify such PMAC as "multilaned" in the ways described in [9]. This would yield a parallelizable construction (which would resist birthday attacks). The only problem with this construction is that it is not truly one-pass. We leave it as an open problem whether we can construct a mode of operation that enjoys all the seven properties in our construction as well as parallelizability.

Reducing the State Size. Our mode is based on three data flows, summing up to $b + n + n = b + 2n$ bits of state size. This is larger than $2n$, the number of bits we expect to be necessary to preclude birthday attacks. It is an interesting problem to see how many of $b + 2n$ bits we can reduce down to $2n$ bits with a new construction in future work.

8 Concluding Remarks

Remarks on Checksum Construction. The idea of message checksum and that of intermediate-value checksum appear in various scenarios, including CBC with Checksum [22,23], 3GPP $f9$ [24] and O-NMAC [25]. The same techniques are also used in the context of keyless hash functions, the purpose being, among other things, to preclude multi-block collision attacks [26]. However, many of these hash functions are broken subsequently after their introduction [27,28].

On the other hand, checksum techniques are proven to be effective (among other things) for extending a distribution property of a compression function to the whole hash function [29]. Our construction presents another positive application of the techniques—providing a secure PRF without the birthday barrier.

Remarks on Masking Technique. The masking technique used in the present work might be contrasted to that in constructing target-collision-resistant (TCR) hash functions [30]. The difference lies in the number of necessary "randomness." In the case of TCR hash functions the construction requires fresh masks as many as logarithmic of the message length (for each message), whereas in our case all the masks are derived from a single mask (which is also derived from a single key) for all messages. Having or not having a "secret" key seems to be essential to making the difference here.

Acknowledgments. The author would like to thank anonymous FSE 2008 reviewers for their valuable comments. In particular, the author wishes to express his appreciation to one of the referees for pointing out an incomplete description of the methods for incrementing masks in an earlier version of the paper. The author is grateful to Kazumaro Aoki for having discussions to fix the problem.

The author received useful feedback from the participants in the FSE 2008 Workshop. Special thanks go to Tetsu Iwata, Kaisa Nyberg, Taizo Shirai and Thomas Shrimpton.

References

1. Bellare, M., Canetti, R., Krawczyk, H.: Keying hash functions for message authentication. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 1–15. Springer, Heidelberg (1996)
2. Bellare, M., Kilian, J., Rogaway, P.: The security of cipher block chaining. In: Desmedt, Y.G. (ed.) CRYPTO 1994. LNCS, vol. 839, pp. 341–358. Springer, Heidelberg (1994)
3. Preneel, B., van Oorschot, P.C.: MDx-MAC and building fast MACs from hash functions. In: Coppersmith, D. (ed.) CRYPTO 1995. LNCS, vol. 963, pp. 1–14. Springer, Heidelberg (1995)
4. Preneel, B., van Oorschot, P.C.: On the security of iterated message authentication codes. IEEE Transactions on Information Theory 45(1), 188–199 (1999)
5. Bellare, M., Guérin, R., Rogaway, P.: XOR MACs: New methods for message authentication using finite pseudorandom functions. In: Coppersmith, D. (ed.) CRYPTO 1995. LNCS, vol. 963, pp. 15–28. Springer, Heidelberg (1995)
6. Jaulmes, É., Joux, A., Valette, F.: On the security of randomized CBC-MAC beyond the birthday paradox limit: A new construction. In: Daemen, J., Rijmen, V. (eds.) FSE 2002. LNCS, vol. 2365, pp. 237–251. Springer, Heidelberg (2002)
7. den Boer, B., Rompay, B.V., Preneel, B., Vandewalle, J.: New (two-track-)MAC based on the two trails of RIPEMD. In: Vaudenay, S., Youssef, A.M. (eds.) SAC 2001. LNCS, vol. 2259, pp. 314–324. Springer, Heidelberg (2001)

8. Lucks, S.: A failure-friendly design principle for hash functions. In: Roy, B. (ed.) ASIACRYPT 2005. LNCS, vol. 3788, pp. 474–494. Springer, Heidelberg (2005)
9. Yasuda, K.: Multilane HMAC—Security beyond the birthday limit. In: Srinathan, K., Rangan, C.P., Yung, M. (eds.) INDOCRYPT 2007. LNCS, vol. 4859, pp. 18–32. Springer, Heidelberg (2007)
10. Black, J., Halevi, S., Krawczyk, H., Krovetz, T., Rogaway, P.: UMAC: Fast and secure message authentication. In: Wiener, M.J. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 216–233. Springer, Heidelberg (1999)
11. Bellare, M., Goldreich, O., Krawczyk, H.: Stateless evaluation of pseudorandom functions: Security beyond the birthday barrier. In: Wiener, M.J. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 270–287. Springer, Heidelberg (1999)
12. Aiello, W., Venkatesan, R.: Foiling birthday attacks in length-doubling transformations – Benes: A non-reversible alternative to Feistel. In: Maurer, U.M. (ed.) EUROCRYPT 1996. LNCS, vol. 1070, pp. 307–320. Springer, Heidelberg (1996)
13. Patarin, J.: Improved security bounds for pseudorandom permutations. In: ACM Conference on Computer and Communications Security, pp. 142–150 (1997)
14. Patarin, J.: About Feistel schemes with six (or more) rounds. In: Vaudenay, S. (ed.) FSE 1998. LNCS, vol. 1372, pp. 103–121. Springer, Heidelberg (1998)
15. Lucks, S.: The sum of PRPs is a secure PRF. In: Preneel, B. (ed.) EUROCRYPT 2000. LNCS, vol. 1807, pp. 470–484. Springer, Heidelberg (2000)
16. Maurer, U.M.: Indistinguishability of random systems. In: Knudsen, L.R. (ed.) EUROCRYPT 2002. LNCS, vol. 2332, pp. 110–132. Springer, Heidelberg (2002)
17. Gligor, V.D., Donescu, P.: Fast encryption and authentication: XCBC encryption and XECB authentication modes. In: Matsui, M. (ed.) FSE 2001. LNCS, vol. 2355, pp. 92–108. Springer, Heidelberg (2002)
18. Black, J., Rogaway, P.: A block-cipher mode of operation for parallelizable message authentication. In: Knudsen, L.R. (ed.) EUROCRYPT 2002. LNCS, vol. 2332, pp. 384–397. Springer, Heidelberg (2002)
19. Jutla, C.S.: Encryption modes with almost free message integrity. In: Pfitzmann, B. (ed.) EUROCRYPT 2001. LNCS, vol. 2045, pp. 529–544. Springer, Heidelberg (2001)
20. Rogaway, P.: Efficient instantiations of tweakable blockciphers and refinements to modes OCB and PMAC. In: Lee, P.J. (ed.) ASIACRYPT 2004. LNCS, vol. 3329, pp. 16–31. Springer, Heidelberg (2004)
21. Brillhart, J., Lehmer, D.H., Selfridge, J.L., Tuckerman, B., Wagstaff Jr., S.S.: Factorizations of $b^n \pm 1$, $b = 2, 3, 5, 6, 7, 10, 11, 12$ Up to High Powers, 3rd edn. Contemporary Mathematics, vol. 22. AMS (2002)
22. Menezes, A.J., van Oorschot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography. CRC Press, Boca Raton (1996)
23. Schneier, B.: Applied Cryptography, 2nd edn. John Wiley, Chichester (1996)
24. 3GPP: Specification of the 3GPP Confidentiality and Integrity Algorithms; Document 1: *f8* and *f9* Specification. 3.1.1 edn (2001)
25. Gauravaram, P., Millan, W., Nieto, J.G., Dawson, E.: 3C – A provably secure pseudorandom function and message authentication code. A new mode of operation for cryptographic hash function. Cryptology ePrint Archive Report 2005/390 (2005)
26. Gauravaram, P., Millan, W., Dawson, E., Viswanathan, K.: Constructing secure hash functions by enhancing Merkle-Damgård construction. In: Batten, L.M., Safavi-Naini, R. (eds.) ACISP 2006. LNCS, vol. 4058, pp. 407–420. Springer, Heidelberg (2006)

27. Joscák, D., Tuma, J.: Multi-block collisions in hash functions based on 3C and 3C+ enhancements of the Merkle-Damgård construction. In: Rhee, M.S., Lee, B. (eds.) ICISC 2006. LNCS, vol. 4296, pp. 257–266. Springer, Heidelberg (2006)
28. Gauravaram, P., Kelsey, J.: Linear-XOR and Additive Checksums Don't Protect Damgård-Merkle Hashes from Generic Attacks. In: Malkin, T. (ed.) CT-RSA 2008. LNCS, vol. 4964, pp. 36–51. Springer, Heidelberg (2008)
29. Lei, D., Li, C.: Extended multi-property-preserving and ECM-construction. In: Srinathan, K., Rangan, C.P., Yung, M. (eds.) INDOCRYPT 2007. LNCS, vol. 4859, pp. 361–372. Springer, Heidelberg (2007)
30. Shoup, V.: A composition theorem for universal one-way hash functions. In: Preneel, B. (ed.) EUROCRYPT 2000. LNCS, vol. 1807, pp. 445–452. Springer, Heidelberg (2000)

Post-Processing Functions for a Biased Physical Random Number Generator

Patrick Lacharme

Imath, Université de Toulon

Abstract. A corrector is used to reduce or eliminate statistical weakness of a physical random number generator. A description of linear corrector generalizing post-processing described by M. Dichtl at FSE'07 [5] is introduced. A general formula for non linear corrector, determining the bias and the minimal entropy of the output of a function is given. Finally, a concrete and efficient construction of post-processing function, using resilient functions and cyclic codes, is proposed.

Keywords: bias, linear correcting codes, Fourier transform, resilient functions, entropy.

1 Introduction

The scheme of a true random number generator consists of two different parts. The first one is a noise source using a physical non deterministic phenomenon producing a raw binary sequence. The second one is a corrector compressing this sequence in order to provide randomness extraction. [1] At FSE'07, M. Dichtl proposed several true random number generators designed to reduce the bias of the noise source and extract more entropy than known algorithms [5]. He considered that the physical source produces statistically independents bits with constant bias. In his conclusion, the author suggested to extend his work in many directions : compression rates, other input sizes and systematic construction of good post-processing functions.

In this paper, we study the output bias of a function. The same assumptions as in [5] are taken : the input bits of the function are independents and have the same bias. General constructions of functions achieving very good output bias are exposed. Furthermore, these functions are very efficiently implemented in smart-card applications. The output bias of a linear corrector is bounded in Section 2, using linear correcting codes. Section 3 presents the explicit calculation of the output bias of a function with its Fourier transform. Resilient functions are used in Section 4 to construct correctors and Section 5 proposes an estimation of minimal entropy of the output sequence.

[1] True random number generator should not be used for cryptographic purposes without a more complex structure as a pseudo random generator [3].

K. Nyberg (Ed.): FSE 2008, LNCS 5086, pp. 334–342, 2008.

2 A Linear Corrector

We consider a physical noise source providing a raw binary sequence. The bits x_i are independent and display a constant bias e, defined by

$$e = \frac{1}{2}(P(x_i = 1) - P(x_i = 0)) \ ,$$

with $P(x_i = 1) = \frac{1}{2} + e$ and $P(x_i = 0) = \frac{1}{2} - e$. This assumption is taken in order to get a simple formula and to compare our correctors with the correctors proposed in [5] on the same hypothesis. Nevertheless Theorem 1 can be generalized with non constant bias assumption.

The linear corrector H proposed in [5], maps 16 bits to 8 bits. For $x = (x_0, \ldots, x_{15})$ the input vector and $y = (y_0, \ldots, y_7)$ the output vector, the corrector H is defined by the following relation

$$\forall i = 0, \ldots, 7 \quad y_i = x_i + x_{i+1 \bmod 8} + x_{i+8} \bmod 2 \ .$$

The compression rate of H is 2, exactly the same rate as the xor corrector

$$y_i = x_{2i} + x_{2i+1} \bmod 2 \ .$$

If we note X_1 and X_2 the two input bytes of H, + the bitwise xor and $RL(X, i)$ the circular rotation of i bits, we can write H in pseudocode

$$H(X_1, X_2) = X_1 + \mathrm{RL}(X_1, 1) + X_2 \ .$$

Two furthers improvements of H are presented

$$H_2(X_1, X_2) = X_1 + \mathrm{RL}(X_1, 1) + \mathrm{RL}(X_1, 2) + X_2 \ ,$$
$$H_3(X_1, X_2) = X_1 + \mathrm{RL}(X_1, 1) + \mathrm{RL}(X_1, 2) + \mathrm{RL}(X_1, 4) + X_2 \ .$$

The author says that if the bias of any input bits is e, then the lowest power of e in the bias of output bytes is 3 for H, 4 for H_2 and 5 for H_3. His approach is to determine probability of every inputs, and to sum up the probability for all input leading to the same output of the corrector.

A simple mathematical proof of previous results is determined using the matricial representation of a linear corrector. For $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_m)$, any linear binary corrector mapping n bits to m bits, is defined as the product of the vector x by the binary matrix $G = (g_{i,j})$:

$$\begin{pmatrix} g_{1,1} \cdots g_{1,n} \\ \vdots \\ g_{m,1} \cdots g_{m,n} \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix} .$$

Theorem 1. *Let G be a linear corrector mapping n bits to m bits. Then the bias of any non zero linear combination of the output bits is less or equal than $2^{d-1}e^d$, where d is the minimal distance of the linear code constructed by the generator matrix G.*

Proof. Firstly, recall that if n bits $x_1, \ldots, x_n$ have a bias e, then the bias of $x_1 + \ldots + x_n \bmod 2$ is $2^{n-1}e^n$ (the proof is a simple induction) [7].

The matrix G is seen as a generator matrix of a $[n, k, d]$ linear code. By definition of the minimal distance of the code, any non zero linear combination of output bits is the sum of, at least, d input bits. We conclude that the bias of any non zero linear combination of output bits is less or equal than $2^{d-1}e^d$. □

This theorem gives an upper bound of the output bias for an arbitrary linear corrector. In particular, the matrix corresponding to H, H_2 and H_3 are respectively generator matrix of $[16, 8, 3]$, $[16, 8, 4]$ and $[16, 8, 5]$ linear codes. Then the bias of any linear combination of output bits is bounded, respectively by $4e^3$, $8e^4$ and $16e^5$. Theorem 5 of Section 5 allows to conclude on Dichtl results on the lowest power of e in the output bytes bias.

Any linear $[n, m, d]$-code provides a linear corrector with an estimation of its output bias. The compression rate of a corrector mapping n bits to m bits is defined by n/m. A table of linear codes gives good linear corrector with variable compression rates and input sizes [6]. The hardware implementation of linear corrector is efficiently achieved as a simple multiplication of an input vector by a constant matrix. A cyclic code provides a more compact implementation of the corrector and improves its realisation.

There are no linear binary codes of length 16, dimension 8 with minimal distance greater than 5 [6]. In theses conditions, to minimize output bias, we must search non linear correctors.

3 Non Linear Corrector

Let f be a corrector mapping n-bits to m-bits. A non zero linear combination of output bits of f is defined using a m-bits vector $u \neq 0$, by the Boolean function $\phi_u(x) = \sum_{i=1}^{m} u_i f_i(x) = u.f(x)$. For an input bits bias e, the bias of this linear combination is

$$\Delta_u = \frac{1}{2}(P(\phi_u(x) = 1) - P(\phi_u(x) = 0)).$$

The bias Δ_u can be directly computed using the truth table of $\phi_u(x)$ and the input bias e by the formula

$$\begin{aligned} 2\Delta_u(e) &= \sum_{\substack{x\in\mathbf{F}_2^n \\ \phi_u(x)=1}} (\frac{1}{2} - e)^{n-w_h(x)}(\frac{1}{2} + e)^{w_h(x)} - \sum_{\substack{x\in\mathbf{F}_2^n \\ \phi_u(x)=0}} (\frac{1}{2} - e)^{n-w_h(x)}(\frac{1}{2} + e)^{w_h(x)} \\ &= \sum_{x\in\mathbf{F}_2^n} (\frac{1}{2} - e)^{n-w_h(x)}(\frac{1}{2} + e)^{w_h(x)}(-1)^{\phi_u(x)+1} . \end{aligned}$$

Therefore

$$\Delta_u(e) = -\frac{1}{2} \sum_{x\in\mathbf{F}_2^n} (\frac{1}{2} - e)^{n-w_h(x)}(\frac{1}{2} + e)^{w_h(x)}(-1)^{\phi_u(x)} . \tag{1}$$

For a Boolean function f, the Hamming weight $w_H(f)$ denotes the number of '1' in its truth table. The Walsh transform of f is :

$$\forall v \in \mathbf{F}_2^n \quad \widehat{f}(v) = \sum_{x \in \mathbf{F}_2^n} (-1)^{f(x)+v.x} \ .$$

Lemma 1. *Let x be a binary vector on $\mathbf{F}_2^n$ such that the bits x_i are independent. Then*

$$\sum_{a \in \mathbf{F}_2^n} P(x=a)(-1)^{v.a} = (-2e)^{w_H(v)} \ .$$

Proof. By independency of the bits x_i,

$$\begin{aligned}\sum_{a \in \mathbf{F}_2^n} P(x=a)(-1)^{v.a} &= \prod_{i=1}^{n} \sum_{a_i=0}^{1} P(x_i=a_i)(-1)^{v_i a_i} \\ &= \prod_{\substack{i=1 \\ v_i=1}}^{n} (P(x_i=0)-P(x_i=1)) \prod_{\substack{i=1 \\ v_i=0}}^{n} (P(x_i=0)+P(x_i=1)) \\ &= (-2e)^{w_H(v)} \ .\end{aligned}$$

Theorem 2 presents a complete description of the bias of any non zero linear combination $\phi_u(x) = u.f(x)$ of a vectorial function f relatively to the the input bias e and the coefficients of the Walsh transform of ϕ_u :

Theorem 2. *Let f be a function which maps n bits to m bits and e the input bit bias. Then the bias $\Delta_u(e)$ is*

$$\Delta_u(e) = \frac{1}{2^{n+1}} \sum_{v \in \mathbf{F}_2^n} (2e)^{w_H(v)} (-1)^{w_H(v)+1} \widehat{\phi_u}(v) \ . \tag{2}$$

Proof. By definition on bias Δ_u,

$$\begin{aligned}2\Delta_u(e) &= P(\phi_u(x)=1) - P(\phi_u(x)=0) \\ &= -\sum_{a \in \mathbf{F}_2^n} P(x=a)(-1)^{\phi_u(a)} \ .\end{aligned}$$

Moreover,

$$\sum_{v \in \mathbf{F}_2^n} (-1)^{v.(a+z)} = \begin{cases} 0 \text{ for } a \neq z \\ 2^n \text{ for } a = z \end{cases} \tag{3}$$

Using equation (3) we get

$$\sum_{a \in \mathbf{F}_2^n} P(x=a)(-1)^{\phi_u(a)} = 2^{-n} \sum_{v \in \mathbf{F}_2^n} \sum_{a \in \mathbf{F}_2^n} (-1)^{v.a} P(x=a) \sum_{z \in \mathbf{F}_2^n} (-1)^{\phi_u(z)+v.z} \ .$$

Therefore with Lemma 1 and definition of $\widehat{\phi_u}$,

$$\Delta_u(e) = \frac{1}{2^{n+1}} \sum_{v \in \mathbf{F}_2^n} (2e)^{w_H(v)} (-1)^{w_H(v)+1} \widehat{\phi_u}(v) \ .$$

□

For example, let f be the quadratic Boolean function defined by

$$f(x) = f(x_1, x_2, x_3) = x_2 + x_3 + x_1x_2 + x_2x_3 \bmod 2 \ ,$$

where the truth table and the Walsh coefficients are

x	$f(x)$	$\widehat{f}(x)$
000	0	0
001	1	4
010	1	0
100	0	-4
011	1	4
101	1	0
110	0	4
111	0	0

The probability $P(f(x) = 0) = \frac{1}{2} - e$, computed using the truth table of f :

$$\begin{aligned} P(f(x) = 0) &= (\frac{1}{2} - e)^3 + (\frac{1}{2} - e)^2(\frac{1}{2} + e) + (\frac{1}{2} - e)(\frac{1}{2} + e)^2 + (\frac{1}{2} + e)^3 \\ &= \frac{1}{2} + 2e^2 \ . \end{aligned}$$

The output bias computed with Theorem 2 gives (with $u = 1$) :

$$\begin{gathered} \Delta_1(e) = \frac{1}{16}(\hat{f}(000) + 2e\hat{f}(001) + 2e\hat{f}(010) + 2e\hat{f}(100) \\ -4e^2\hat{f}(011) - 4e^2\hat{f}(101) - 4e^2\hat{f}(110) + 8e^3\hat{f}(111)) \end{gathered}$$

After reduction, we get

$$\Delta_1(e) = -2e^2 \ .$$

Definition 1. *Let P be a polynomial of degree d, defined by*

$$P(X) = \sum_{i=0}^{d} a_i X^i \ .$$

The valuation of P is the minimal $i > 0$ such that $a_i \neq 0$.

Corollary 1 is a consequence of Theorem 2 :

Corollary 1. *Let f be a function mapping n bits to m bits and e the input bias of the function. For any vector u, we define for all w, with $0 \leq w \leq n$*

$$B_w = \sum_{\substack{v \in \mathbf{F}_2^n \\ w_H(v) = w}} \widehat{\phi_u}(v) \ .$$

Then the bias of $\phi_u(x)$ is a polynomial of valuation W, with

$$W = \min\{w \mid B_w \neq 0\} \ .$$

Formula (2) gives a complete description of the bias and coefficients of the polynomial $\Delta_u(e)$ are determined by B_w.

In particular, if we consider the linear Boolean function which is the sum of d variables, then $B_w = 0$ for all $w \neq d$.

4 A Resilient Corrector

A (n, m, t)-resilient function is a function mapping n bits to m bits such that if t input bits are fixed, there is no influence on the output :

Definition 2. *[4] A (n, m, t)-resilient function is a function f mapping $\mathbf{F}_2^n$ to $\mathbf{F}_2^m$ such that for any coordinates $i_1, \dots i_t$ and for any binary constant $c_1, \dots, c_t$ and for all $y \in \mathbf{F}_2^m$, we have*

$$P(f(x) = y \mid x_{i_1} = c_1, \dots, x_{i_t} = c_t) = 2^{-m},$$

where x_i with $i \notin \{i_1, \dots, i_t\}$ verify $P(x_i = 1) = P(x_i = 0) = 0.5$.

A (n, m, t)-linear resilient function is a linear corrector[2] and Theorem 3 shows the relation between resilience degree of a linear function and output bias :

Lemma 2. [4] *A $(n \times m)$ binary matrix M is a generator matrix of a linear $[n, m, d]$-code if and only if the function*

$$x \mapsto M.{}^t x$$

is a linear $(n, m, d-1)$-resilient function.

Theorem 3. *Let f be a linear (n, m, t)-resilient function. Then the bias of any non zero linear combination of the output bits is less or equal than $2^t e^{t+1}$.*

Proof. From Lemma 2, any linear (n, m, t)-resilient function provides a generator matrix of a $[n, m, t+1]$-linear code. The theorem follows with Theorem 1. □

In the case of the (n, m, t)-resilient function is non linear, Theorem 4 evaluates the valuation of the output bias, using the resilience order of the function. Lemma 3 is known as xor Lemma [4]:

Lemma 3. *Let f be a (n, m, t)-resilient function and u a non zero vector in $\mathbf{F}_2^m$. Then, any non zero linear combination $u.f(x)$ of f is a $(n, 1, t)$-resilient Boolean function.*

G. Xiao and J. Massey propose a spectral characterization of (n, m, t)-resilient functions [11] :

Lemma 4. *Let f be a $(n, 1, t)$-resilient Boolean function. Then for all vector v in $\mathbf{F}_2^n$ with $w_H(v) \leq t$, we have $\widehat{f}(v) = 0$.*

[2] In [9], Stinson, Martin and Sunar have proposed a true random number generator using a linear resilient function for the post-processing.

Theorem 4. *Let f be (n, m, t)-resilient function and all input bits have a bias e. Then the bias of any non zero linear combination of output bits is a polynomial in e of valuation greater than $t + 1$.*

Proof. Let $\phi_u(x) = u.f(x)$ be a linear combination of output bits. By Lemma 3 ϕ_u is a $(n, 1, t)$-resilient Boolean function. So, all Walsh coefficients $\widehat{\phi_u}(v)$ are null for all vector v of Hamming weight less or equal than t (Lemma 4). Using Theorem 2, we get

$$\Delta_u(e) = \frac{1}{2^{n+1}} \sum_{\substack{v \in \mathbf{F}_2^n \\ w_H(v) > t}} (2e)^{w_H(v)} (-1)^{w_H(v)+1} \widehat{\phi_u}(v) \ .$$

□

In the non linear case, the resilient property is not a necessary condition to reduce the bias. The Boolean function of the previous example

$$f(x) = x_2 + x_3 + x_1 x_3 + x_2 x_3 \bmod 2 \ ,$$

is not resilient, but the output bias is reduced. Indeed, the Walsh coefficients of (001) and (100) are not null, but the sum of both is null.

For example, M. Dichtl proposed a non linear corrector mapping 16 bits to 8 bits such that all e powers up to the fifth are gone in the output bias formula [5]. This corrector was found by exhaustive search and the hardware implementation requires a considerable amount of chip area.

The calculation of syndrome of the non linear $(16, 256, 6)$ Nordstrom-Robinson code provides a $(16, 8, 5)$-resilient function [10]. Theorem 4, applied to this function, gives a corrector with a valuation of $\Delta_u(e)$ equal to 6 and with a possible implementation for smart-card applications.

5 Bias and Minimal Entropy

For the evaluation of the random quantity in a binary sequence, the minimal entropy is an appropriate notion for random number generation in cryptography [3]. In this part, we prove that if the bias of any non zero linear combination of output bits is bounded, then the minimal entropy of the output can be estimated. Theorem 5 gives the relation between one-dimensional bias $\Delta_u(e)$ and multidimensional bias and follows from [1], [2].

Theorem 5. *Let f be a function from $\mathbf{F}_2^n$ to $\mathbf{F}_2^m$. For all $y \in \mathbf{F}_2^m$, the multidimensional bias*

$$\left| P(f(x) = y) - 2^{-m} \right|$$

is less or equal than

$$2 \max_{u \in \mathbf{F}_2^m} |\Delta_u| \ .$$

Definition 3. *Let X be a discrete random variable on $\{0,1\}^n$. The minimal entropy of X is the maximal number k such that*

$$\forall x \in X, \quad P(X = x) \leq 2^{-k} \ .$$

Theorem 5 is a good tool to evaluate minimal entropy of the output. Indeed, we suppose that a (n, m, t)-resilient function is used, with an input bias e. Then, with Theorems 4 and 5, the bias of any output m-tuple is a polynomial of valuation greater than $t + 1$. If e^{t+1} is negligible compared to 2^{-m}, then the minimal entropy of the output is very close to m.

With a linear (n, m, t)-resilient function and an input bias e, we have

$$P(f(x) = y) \leq 2^{-m} + 2^{t+1} e^{t+1} \ ,$$

then the minimal entropy of the output is greater than

$$m - \log_2(1 + e^{t+1} 2^{m+t+1}) \ .$$

For example, if $e = 1/4$, then

$$P(f(x) = y) \leq 2^{-m} + 2^{-(t+1)} \ ,$$

For a linear cyclic code, a syndrome is computed with a modular polynomial reduction, which is realized by using a linear feedback shift register. Lemma 2 explains how getting linear resilient function by calculating a syndrome. Let C a $[n, k, d]$ linear code, H its check matrix and d' its dual distance, then the function $x \mapsto H.{}^t x$ is a $(n, n-k, d'-1)$-resilient function.

Let C the $[255, 21, 111]$ BCH code, D the $[255, 234, 6]$ dual code of C, with generator polynomial

$$H(X) = X^{21} + X^{19} + X^{14} + X^{10} + X^7 + X^2 + 1 \ .$$

The input 255-tuple $(m_1, \ldots, m_{255})$ is coded by a binary polynomial $m(X) = \sum_{i=1}^{255} m_i X^i$. Therefore the function f mapping $\mathbf{F}_2^{255}$ to $\mathbf{F}_2^{21}$, defined by

$$m(X) \mapsto m(X) \bmod H(X)$$

is a $(255, 21, 110)$-resilient function. This polynomial reduction is implemented by a shift register of length 21 with only seven xor gates.

In this case, with an important input bias $e = 0.25$, Theorems 3 and 5 give an output bias of :

$$\forall y \in \mathbf{F}_2^{21} \quad \left| P(f(X) = y) - 2^{-21} \right| \leq 2^{-111} \ .$$

Therefore, the minimal entropy of the output is very close to 21.

6 Conclusion

In this work we present general constructions of good post-processing functions. We have shown that linear correcting codes and resilient functions provide many correctors achieving good bias reduction with variable input sizes. Linear feedback shift register are suitable for an hardware inplementation where the chip area is limited.

Acknowledgment

The author would like to thank Philippe Langevin for helpful discussions. The author want also thank to Kaisa Nyberg for useful comments on the paper and for the proof of Theorem 2, which is more elegant than original proof presented at the workshop.

References

1. Alon, N., Goldreich, O., Hastad, J., Peralta, R.: Simple Constructions of Almost k-wise Independent Random Variables. In: IEEE Symposium on Foundations of Computer Science, pp. 544–553., http://citeseer.ist.psu.edu/alon92simple.html
2. Baignères, T., Junod, P., Vaudenay, S.: How far can we go beyond linear cryptanalysis. In: Lee, P.J. (ed.) ASIACRYPT 2004. LNCS, vol. 3329, pp. 432–450. Springer, Heidelberg (2004)
3. Barker, E., Kelsey, J.: Recommendation for random number generation using deterministic random bit generators (revised). NIST Special publication 800-90 (March 2007), http://csrc.nist.gov/publications/nistpubs/800-90/SP800-90revised_March2007.pdf
4. Chor, B., Goldreich, O., Hastad, J., Freidmann, J., Rudich, S., Smolensky, R.: The bit extraction problem or t-resilient functions. In: Proc. 26th IEEE Symposium on Foundations of Computer Sciences, pp. 396–407 (1985), http://citeseer.ist.psu.edu/chor85bit.html
5. Dichtl, M.: Bad and good ways of post-processing biased physical random numbers. In: Biryukov, A. (ed.) FSE 2007. LNCS, vol. 4593, pp. 127–152. Springer, Heidelberg (2007)
6. Grassl, M.: Code table: bounds on the parameters of various types of codes, http://www.codestables.de
7. Matsui, M.: Linear cryptanalysis method of DES Cipher. In: Helleseth, T. (ed.) EUROCRYPT 1993. LNCS, vol. 765, pp. 386–397. Springer, Heidelberg (1994)
8. Mac Williams, F.J., Sloane, N.J.A.: The theory of error correcting codes. North-Holland, Amsterdam (1977)
9. Martin, W.J., Sunar, B., Stinson, D.R.: A provably secure true random number generator with built in tolerance to active attacks. IEEE Transactions on computers 56(1), 109–119 (2007)
10. Stinson, D.R., Massey, J.: An infinite class of counterexamples to a conjecture concerning non linear resilient functions. Journal of cryptology 8(3), 167–173 (1995), http://citeseer.ist.psu.edu/629195.html
11. Xiao, G.: Massey: A spectral Characterization of correlation immune functions. IEEE Transactions on information theory V 34, 569–571 (1988)

Entropy of the Internal State of an FCSR in Galois Representation

Andrea Röck

Team SECRET, INRIA Paris-Rocquencourt, France
andrea.roeck@inria.fr
http://www-rocq.inria.fr/secret/

Abstract. Feedback with Carry Shift Registers (FCSRs) are primitives that are used in multiple areas like cryptography or generation of pseudorandom sequences. In both cases, we do not want that an attacker can easily guess the content of the register. This requires a high entropy of the inner state. We consider the case of a binary FCSR in Galois representation. In this article, we show that we already lose after one iteration a lot of entropy. The entropy reduces until the moment where the FCSR reaches a periodic behavior. We present an algorithm which computes the final entropy of an FCSR and which also allows us to show that the entropy never decreases under the size of the main register.

1 Introduction

FCSRs are finite state machines which were independently introduced by Goresky and Klapper [KG93, KG97], Marsaglia and Zamand [MZ91], and Couture and L'Ecuyer [CL94]. They are similar to Linear Feedback Shift Registers (LFSRs). However, they use an additional register to store the carry information and their transition function is non linear, more precisely quadratic [AB05b]. Goresky and Klapper [GK02] distinguish between FCSRs in Fibonacci and Galois representation. In this article, we consider binary FCSRs in Galois architecture.

An application of FCSRs is cryptography, as for example the stream cipher presented by Arnaut and Berger in [AB05b]. In this area, the entropy is an important parameter. It represents the minimal number of binary questions an attacker has to ask on average to obtain an unknown value. Therefore, we are always interested in a high entropy. In the following, we study the entropy of the inner state of an FCSR. First we give some notations. Subsequently, we will present the structure of an FCSR in the Galois representation and explain exactly the meaning of the state entropy. In the end, we give an overview of this article.

1.1 Notations

In this article, we are going to use the following notations to describe the behavior of an FCSR.

n: Let n denote the size of the main register M in bits.

K. Nyberg (Ed.): FSE 2008, LNCS 5086, pp. 343–362, 2008.

m: We mean by m the 2-adic description of the current state of M: $m = \sum_{i=0}^{n-1} m_i\, 2^i$, where $m_0, \ldots, m_{n-1}$ are the bits in M. Thus, $0 \leq m < 2^n$.

d: Let d be an integer with $2^{n-1} \leq d < 2^n$. We will use it to determine at which positions we have a feedback with carry. We mean by d_i the i'th bit of the binary representation of d. We define $d_i = 0$ for all $i < 0$.

$I_d = \{0 \leq i \leq n-2 | d_i = 1\}$: The set I_d contains all feedback positions.

$d^* = d - 2^{n-1}$.

$\ell = wt(d^*)$: We denote with ℓ the number of feedback branches, where $wt(d^*)$ means the hamming weight of d^*, *i.e.* the number of 1's in its binary representation.

$c = \sum_{i \in I_d} c_i\, 2^i$: The value c is the 2-adic description of the carry bits.

$(m(t), c(t))$: The pair $(m(t), c(t))$ represents the actual value of the state after t iterations.

$q = 1-2d$: We denote by q the divisor in the 2-adic description of the produced bit string. It holds that $q < 0$.

$p = m + 2c$: The value p represents the corresponding dividend which can be in the range of $0 \leq p \leq |q|$.

s: Let s is the binary representation of the output sequence of the generator.

$H(1)$: Let $H(1)$ denote the entropy of the state after one iteration.

H^f: As soon as the FCSR obtains a periodic behavior, the state entropy does not change any more. We denote this final entropy by H^f.

Furthermore, we need two special sums in our calculations:

$$S_1(k) = \sum_{x=2^{k-1}+1}^{2^k} x \log_2(x),$$

$$S_2(k) = \sum_{x=1}^{2^k-1} x \log_2(x).$$

1.2 FCSR in Galois

A binary FCSR in Galois representation is built like a LFSR with a main register and several feedback branches. However, in the case of the FCSRs we have an additional carry bit at each feedback position. An example for such an FCSR is presented in Fig. 1.

To change the state from $(m(t), c(t))$ to $(m(t+1), c(t+1))$, we use the following equations:

$$\begin{aligned} m_{n-1}(t+1) &= m_0(t), && (1)\\ i \in I_d : m_i(t+1) &= (m_0(t) + c_i(t) + m_{i+1}(t)) \bmod 2, && (2)\\ c_i(t+1) &= (m_0(t) + c_i(t) + m_{i+1}(t)) \div 2, && (3)\\ i \notin I_d : m_i(t+1) &= m_{i+1}(t), && (4) \end{aligned}$$

where $x \div 2$ means the integer division $\lfloor x/2 \rfloor$. The bit $m_0(t)$ is directly shifted to position $m_{n-1}(t+1)$. In the cases without a feedback, *i.e.* $i \notin I_d$, the bit gets

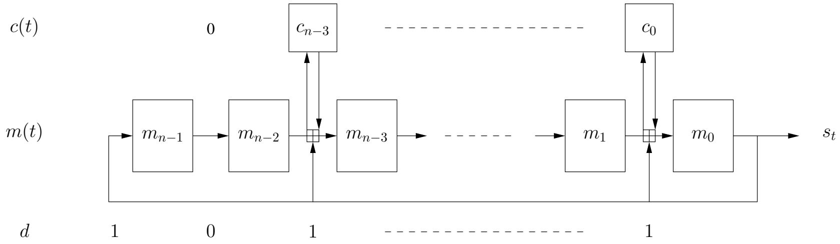


Fig. 1. Model of an FCSR

simply shifted one position to the left. Otherwise, we add the bit in the main register at position $i+1$ with the carry bit and the bit from position 0. This sum modulo 2 is put into the main register. The value of the sum divided by 2 is given back to the carry bit. The equations (2) and (3) can also be written together as:

$$m_i(t+1) + 2c_i(t+1) = m_0(t) + c_i(t) + m_{i+1}(t). \tag{5}$$

Remark 1. The + in all these equations represents the normal integer addition (and not an addition modulo 2) even if we are only using bit values for $m_i(t)$ and $c_i(t)$.

The output sequence of the FCSR can be easily described by means of the feedback positions (determined by d) and the initial state (m, c). Let q and p be defined as $q = 1 - 2d$ and $p = m + 2c$. This means that $q < 0$ and $0 \leq p \leq |q|$. In this case, the output sequence of the FCSR is $s = \frac{p}{q}$ [GK02]. Another property shown in [AB05a] is: Let $(m(t+1), c(t+1))$ be the state produced by $(m(t), c(t))$ after one iteration. Let $p(t) = m(t) + 2c(t)$ and $p(t+1) = m(t+1) + 2c(t+1)$ be the corresponding values of p. Then it holds that:

$$2p(t+1) = p(t) \pmod{q}. \tag{6}$$

where all bits are 0 or 1 respectively.

We can use the following property of an FCSR to determine the period of a sequence: if q is odd, p and q are coprime and $s = p/q$ then the period of s is the order of 2 modulo q, this means the smallest t such that $2^t = 1 \pmod{q}$. If p and q are not coprime, we divide them both by their greatest common divisor. It is easy to see that the maximal period is $|q| - 1$. There are always two fix points $(0, 0)$ and $(2^n - 1, d^*)$ with an period of length 1, which represents the cases where all bits are respectively 0 or 1. It is known [GK02] that if 2 is primitive modulo q, which means it has order $|q| - 1$, then all the other periodic states are obtained by the state $(1, 0)$ by iterating the FCSR. Since q is odd, this implies that, except for the two fix points, we have an FCSR with one single cycle of maximal length $|q| - 1$. Such a sequence is called ℓ–sequence.

1.3 State Entropy

For any discrete probability distribution $P = \{p_1, \ldots, p_Z\}$, Shannon's entropy is defined by:

$$H = \sum_{j=1}^{Z} p_j \log_2 \left(\frac{1}{p_j} \right) . \tag{7}$$

If $p_j = 0$, we use the classical convention in the computation of the entropy: $0 \log_2(\frac{1}{0}) = 0$. This can be done, since a zero probability has no influence in the computation of the entropy. The state of our FCSR consists of $n + \ell$ bits. Let us assume that each initial state is chosen with the same probability $p_{(m(0),c(0))} = 2^{-n-\ell}$. Then by using (7) we know that the initial entropy is $n + \ell$ bits.

Let us consider the update function of an FCSR. It consists of trees of different length, where each root of a tree is a node in a cycle. This is the same for any finite function which is not a permutation. An example of such a graph can be found in Fig. 2. Each node in the graph represents a possible state of the FCSR and each arrow represents an update from one state to another. Each time when a state is produced by more than one state, the number of possible states, and therefore the entropy of the state, reduces. As soon as we reached a point on the cycle from any possible starting points, the FCSR behaves like a permutation and the entropy stays constant. We will denote this value by the final entropy H^f.

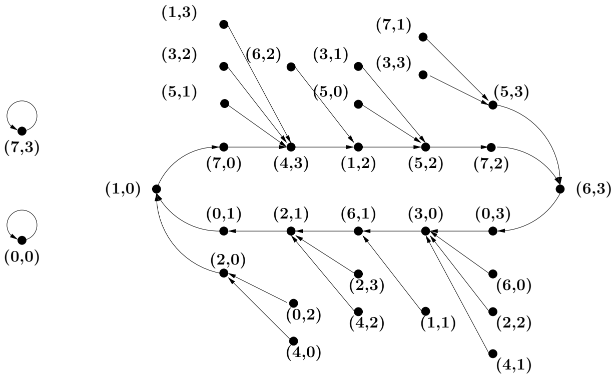


Fig. 2. Functional graph of the FCSR with $n = 3$ and $q = -13$

We need the probability of each state to compute the entropy after some iterations of the update function. If a state is produced by exactly r other states after k iterations, then its probability is $r\ 2^{-n-\ell}$. Using this probability, we can compute the entropy applying (7).

Remark 2. We consider the case where all the $2^{n+\ell}$ different values for the initial state can be chosen, like in the first version of the F-FCSR-8 [ABL05]. This is not always the case, *e.g.* for later versions of the F-FCSR [ABL06], the carry bits of the initial value are always set to 0. This implies that at the beginning this stream cipher has only n bits of entropy, however, it will not lose any more entropy, as we will see later.

1.4 Outline

In Section 2, we compute the state entropy after 1 iteration. Subsequently, in Section 3, we present an algorithm which computes the final entropy for any arbitrary FCSR. This algorithm uses some sums which gets difficult to compute for large ℓ. However, we give a method to compute very close upper and lower bounds of the entropy. The same algorithm is used in Section 4 to prove that the final entropy is always larger than n. We conclude the article in Section 5.

2 Entropy after One Iteration

If the initial value of the state is chosen uniformly, we have an initial state entropy of $n + \ell$ bits. We are interested in how many bits of entropy we already lose after one iteration.

Let us take an arbitrary initial state $(m(0), c(0))$ which produces the state $(m(1), c(1))$ after one iteration. To compute the probability of $(m(1), c(1))$, we want to know by how many other initial states $(m'(0), c'(0)) \neq (m(0), c(0))$ it is produced. From (1), we see that for such an initial state $m'_0(0) = m_{n-1}(1) = m_0(0)$ is fixed. In the same way, we see from (4) that for all $i \notin I_d$ the values $m'_{i+1}(0) = m_i(1) = m_{i+1}(0)$ are already determined. By using (5) and the previous remarks, we can write for $i \in I_d$:

$$\begin{aligned} m_i(1) + 2c_i(1) &= m_0(0) + c'_i(0) + m'_{i+1}(0) \\ m_i(1) + 2c_i(1) &= m_0(0) + c_i(0) + m_{i+1}(0) \end{aligned}$$

and thus,

$$c'_i(0) + m'_{i+1}(0) = c_i(0) + m_{i+1}(0).$$

If $m_{i+1}(0) = c_i(0)$ it must hold that $m_{i+1}(0) = c'_i(0) = m'_{i+1}(0)$ since each value can only be either 0 or 1. Therefore, the only possibility for $(m'(0), c'(0))$ to differ from $(m(0), c(0))$ is that there is a position $i \in I_d$ with $m_{i+1}(0) \neq c_i(0)$.

Let j be the number of positions in the initial state where $i \in I_d$ and $c_i(0) + m_{i+1}(0) = 1$. Then, there are exactly $2^j - 1$ other initial states which produce the same state after one iteration. Thus, $(m(1), c(1))$ has a probability of $\frac{2^j}{2^{n+\ell}}$. We look now how many states $(m(1), c(1))$ have this probability. Such a state must be created by an initial state $(m(0), c(0))$ which has j positions $i \in I_d$ with $c_i(0) + m_{i+1}(0) = 1$. There are $\binom{\ell}{j}$ possibilities to choose these positions. At the remaining $\ell - j$ positions with $i \in I_d$, we have $m_{i+1}(0) = c_i(0) \in \{0, 1\}$. In the same way, we can choose between 0 and 1 for the remaining $n-\ell$ positions. There exists exactly $2^{n-j}\binom{\ell}{j}$ different states $(m(1), c(1))$ with a probability of $2^{j-n-\ell}$.

Using (7), $\sum_{j=0}^{\ell} \binom{\ell}{j} = 2^\ell$ and $\sum_{j=0}^{\ell} j \binom{\ell}{j} = \ell 2^{\ell-1}$ we can write the entropy after one iterations as:

$$\begin{aligned} H(1) &= \sum_{j=0}^{\ell} 2^{n-j} \binom{\ell}{j} 2^{j-n-\ell}(n+\ell-j) \\ &= n + \frac{\ell}{2}. \end{aligned}$$

We have shown that the entropy after one iteration is:

$$H(1) = n + \frac{\ell}{2} \tag{8}$$

which is already $\ell/2$ bits smaller than the initial entropy.

3 Final State Entropy

We have shown that after one iteration the entropy has already decreased by $\ell/2$ bits. We are now interested in down to which value the entropy decreases after several iterations, *i.e.* the final entropy H^f. For the computation of H^f, we need to know how many initial states arrive at the same cycle point after the same number of iterations. We are going to use the following proposition.

Proposition 1. *[ABM08, Prop. 5] Two states (m, c) and (m', c') are equivalent,* i.e. *$m + 2c = m' + 2c' = p$, if and only if they eventually converge to the same state after the same number of iterations.*

Let us assume that we have iterated the FSCR sufficiently many times that we are on the cycle of the functional graph. In this case, we do not have any more collisions. If a state in the cycle is reached by x other states, it has a probability of $x/2^{n+\ell}$. After one iteration, all probabilities shift one position in the direction of the cycle, which corresponds to a permutation of the probabilities. However, the definition of the entropy is invariant to such a permutation. Let $v(p)$ denote the number of states which produce the same p. From Proposition 1, we know that we find a corresponding state in the cycle, which is reached by $v(p)$ states and has a probability of $v(p)/2^{n+\ell}$. We can write the final entropy by means of equation (7):

$$H^f = \sum_{p=0}^{|q|} \frac{v(p)}{2^{n+\ell}} \log_2 \left(\frac{2^{n+\ell}}{v(p)} \right). \tag{9}$$

Remark 3. Let us have a look at an FCSR as mentioned in Remark 2, which always sets $c(0) = 0$. For each $0 \leq p < 2^n$ there is only one possibility to form $p = m$. This means that there are no collision and the final entropy is the same as the initial entropy, namely n bits.

The numerator p can take any value between 0 and $2^n - 1 + 2(d - 2^{n-1}) = 2d - 1 = |q|$. We look at the binary representations of $m, 2c$ and p to find all

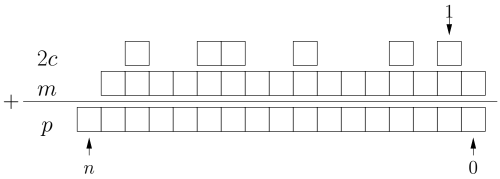


Fig. 3. Evaluation of $p = m + 2c$ bit per bit

possible pairs (m, c) which correspond to a given p. We study bit per bit which values are possible. This idea is presented in Fig. 3, where each box represents a possible position of a bit. We mean by $2c$ the carry bits of the value c all shifted one position to the left.

Remark 4. Due to this shift, we will normally consider the index $i-1$ for d or c.

3.1 Notations

Before continuing we give some additional notations:

$ca(j) = m_j + c_{j-1} + ca(j-1) \pmod 2$: In the following, we consider only the addition $m + 2c$, thus, if we talk of a carry we mean the carry of this integer addition. We mean by $ca(j)$ the carry which is produced by adding the bits m_j, c_{j-1} and the carry of the previous position. *E.g.* if we have $m = 13$ and $c = 2$ we have $ca(1) = 0$ and $ca(2) = ca(3) = 1$. The value $ca(0)$ is always 0 since we only have m_0 for the sum.

$i := \lfloor \log_2(p) \rfloor$: For $1 \leq p \leq |q|$, let i be the index of the most significant bit in p which is not equal to 0.

$\ell' = \#\{j \leq i | d_{j-1} = 1\}$: We define by ℓ' the number of indices smaller or equal to i for which $d_{j-1} = 1$.

$r(p) = max\{j < i | d_{j-1} = 0, p_j = 1\}$: For a given p, let $r(p)$ be the highest index smaller than i such that $d_{j-1} = 0$ and $p_j = 1$. In this case, the carry of the integer addition $m + 2c$ cannot be forwarded over the index $r(p)$. If there is no index j with $d_{j-1} = 0$ and $p_j = 1$, we set $r(p) = -1$. For the case $i < n$ and $d_{i-1} = 0$, we get a range of $-1 \leq r(p) < i$. However, we will see that for $2^n \leq p \leq |q|$, the value $r(p)$ is only possible for $-1 \leq r(p) < log_2(d^*) + 1$. For simplicity reasons, we sometimes write only r if it is clear which p we are meaning or if there are multiple p's with the same value for $r(p)$.

$f_1(r)$: This is a helping function which is needed in the further computations. It is defined as:

$$f_1(r) = \begin{cases} 2^r & \text{for } r \geq 0 \\ 1 & \text{for } r = -1. \end{cases}$$

$\ell''(r) = \#\{j < r | d_{j-1} = 1\}$: For a given r, we define ℓ'' as the number of indices strictly smaller than r for which $d_{j-1} = 1$. Again, we use sometimes only ℓ'' if it is clear to which r we refer.

$v(p) = \#\{(m,c)|m+2c=p\}$: Let $v(p)$ denote the number of pairs (m,c) which create $p = m + 2c$.

In Case 2 in Section 3.2, we will see that it is sufficient to consider the indices j with $r(p) < j < i$ for knowing if there is a carry at position $i-1$. To facilitate the computation, we use the following notations:

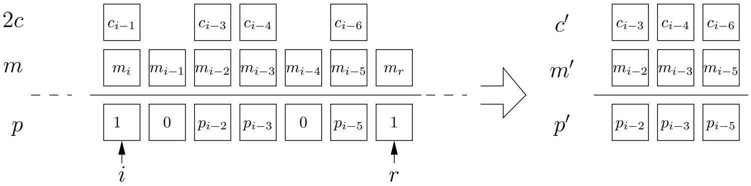


Fig. 4. Reduction of $p, m, 2c$ to p', m', c'

p',m' and c': We mean with p', m' and c' the bit strings $p, m, 2c$ reduced to the positions j with $r < j < i$ and $d_{j-1} = 1$. An example can be seen in Fig. 4.

Remark 5. In the case of c', we do not consider c but $2c$. Let j_1 be an index $r < j_1 < i$ with $d_{j_1-1} = 1$ and j_2 its corresponding index in the reduction, *i.e.* $m'_{j_2} = m_{j_1}$. Then we use $c'_{j_2} = c_{j_1-1}$. Furthermore, the value of c' is a continuous bit string, since we are only interested in positions where there is a feedback register.

The length of p', m' and c' is $\ell' - \ell'' - 1$ bits.

$0p'$ and $1p'$: We obtain $0p'$ and $1p'$ by concatenating respectively 0 and 1 to the left of the bit string p'.

$X(p')$: We denote by $X(p')$ the number of possibilities for m' and c' such that $1p' = m' + c'$, which means that we have a carry at the position of the most significant bit of m'.

3.2 Final Entropy Case by Case

We cannot write the sum (9) for the final entropy directly in a closed form. However, we partition the set of all possible $0 \leq p \leq |q|$ in four cases. For each case, we will evaluate the value $\frac{v(p)}{2^{n+\ell}} \log_2 \left(\frac{2^{n+\ell}}{v(p)} \right)$ for all its p's. We obtain the final sum of the entropy by summing up all these values.

Case 1: $1 \leq i < n$ and $d_{i-1} = 0$

To create $p_i = 1$ at position i, we have to add $m_i + ca(i-1)$. For each value of $ca(i-1)$ there exists exactly one possibility for m_i. For each position j with a feedback bit, $d_{j-1} = 1$, we have two possibilities to create the value of p_j. In this case, we can write for each p:

$$v(p) = 2^{\ell'} .$$

For each i, all p's within the range $[2^i, 2^{i+1}[$ are possible. So we must add:

$$H_1(n,i,\ell,\ell') = 2^i\, 2^{\ell'-n-\ell}(n+\ell-\ell') \tag{10}$$

to the entropy for each $1 \le i \le n$ with $d_{i-1} = 0$.

Case 2: $1 \le i < n$ and $d_{i-1} = 1$:

For a given p we know from the definition of $r(p)$ that for all j's with $r(p) < j < i$, if $d_{j-1} = 0$, then $p_j = 0$. In the case of $(d_{j-1} = 0,\, p_j = 0)$, a carry is always forwarded. This means that for $m + 2c$, if we have $ca(j-1) = 1$, m_j must be 1 and we have $ca(j) = 1$. However, with $ca(j-1) = 0$ we have $m_j = 0$, and so we have $ca(j) = 0$ as well. It is sufficient to consider the $\ell' - \ell'' - 1$ positions j with $i > j > r(p)$ for which $d_{j-1} = 1$, to know if we have a carry at index $i-1$.

From the definition of this case, we know that $p_i = 1$ and $d_{i-1} = 1$. If we have $ca(i-1) = 0$ we have two possibilities for (m_i, d_{i-1}), namely $(1,0)$ and $(0,1)$, to generate the $p_i = 1$. Otherwise, we only have one possibility $(0,0)$. For a given p' we have:

$$X(p') + 2(2^{\ell'-\ell''-1} - X(p')) = 2^{\ell'-\ell''} - X(p') \tag{11}$$

possibilities to choose m',c' and (m_i, d_{i-1}). The following lemma helps us to compute $X(p')$. Its proof is given in Appendix B.

Lemma 1. *Let p', m' and c' be three bit strings of length K. For all $0 \le x \le 2^K - 1$, there exists* exactly one *p' with $X(p') = x$,* i.e. *there exists exactly one p' such that for x different pairs (m', c') we can write $1p' = m' + c'$.*

In our case: $K = \ell' - \ell'' - 1$. The next question is, how many p's have the same reduction p'. If $0 \le r < i$ we have 2^r possibilities, in the case $r = -1$ we have only one. We consider this behavior by using the helping function $f_1(r)$. By combining Lemma 1 and (11), we obtain that for each $2^{\ell'-\ell''-1} + 1 \le y \le 2^{\ell'-\ell''}$ there is exactly one p' which is generated by y different combinations of m' and c'. Each p which corresponds to such a p', is generated by $y\ 2^{\ell''}$ different pairs (m, c), since at each position $j < r$ with $d_{j-1} = 1$ we have two possibilities to create p. This means that this p has a probability of $\frac{y\,2^{\ell''}}{2^{n+\ell}}$. For fixed values of i, r, ℓ' and ℓ'' we have to add the following value to the entropy:

$$\begin{aligned} &H_2(n,r,\ell,\ell',\ell'') \\ &= f_1(r) \sum_{y=2^{\ell'-\ell''-1}+1}^{2^{\ell'-\ell''}} \frac{y2^{\ell''}}{2^{n+\ell}} \log_2\left(\frac{2^{n+\ell}}{y2^{\ell''}}\right) \\ &= f_1(r) 2^{-n-\ell}\left[2^{\ell'-2}\left(3\ 2^{\ell'-\ell''-1}+1\right)(n+\ell-\ell'') - 2^{\ell''} S_1(\ell'-\ell'')\right]. \end{aligned}$$

Thus, in this case, we have to add for every $1 \le i \le n-1$ with $d_{i-1} = 1$, and every $-1 \le r < i$ with $d_{r-1} = 0$ the value:

$$H_2(n,r,\ell,\ell',\ell'') = f_1(r)2^{-n-\ell}\left[2^{\ell'-2}\left(3\,2^{\ell'-\ell''-1} + 1\right)(n+\ell-\ell'') - 2^{\ell''}S_1(\ell'-\ell'')\right] \tag{12}$$

where $\ell' = \ell'(i)$ and $\ell'' = \ell''(r)$.

Case 3: $i = n$, $2^n \leq p \leq |q|$:

In this case, we always need a carry $ca(n-1)$ to create $p_n = 1$. Like in the previous case, we are going to use $r(p)$, ℓ'', p', (m', c') and $X(p')$. However, this time we have $i = n$ and $\ell = \ell'$.

For $p = |q|$, which means that (m, c) consists of only 1's, it holds that for all $n > j > \log_2(d^*) + 1$, we have $p_j = 0$. If we would have a $r(p) \geq \log_2(d^*) + 1$, then p would be greater than $|q|$ which is not allowed. Therefore, r must be in the range of $-1 \leq r < \log_2(d^*) + 1$.

From Lemma 1, we know that for all $1 \leq x \leq 2^{\ell-\ell''} - 1$ there exists exactly one p' with $X(p') = x$, *i.e.* there are x pairs of (m', c') with $1p' = m' + c'$. We exclude the case $X(p') = 0$, because we are only interested in p's that are able to create a carry. For each p', there are $f_1(r)$ possible values of p which are reduced to p'. If p' is created by x different pairs of (m', c'), then each of its corresponding values of p is created by $x\ 2^{\ell''}$ pairs of (m, c) and has a probability of $\frac{x\ 2^{\ell''}}{2^{n+\ell}}$.

For a given r and ℓ'' the corresponding summand of the entropy is:

$$\begin{aligned}
H_3(n, r, \ell, \ell'') &= f_1(r) \sum_{x=1}^{2^{\ell-\ell''}-1} \frac{x\ 2^{\ell''}}{2^{n+\ell}} \log_2\left(\frac{2^{n+\ell}}{x\ 2^{\ell''}}\right) \\
&= f_1(r) 2^{-n} \left[2^{-1}\left(2^{\ell-\ell''} - 1\right)(n + \ell - \ell'') - 2^{\ell''-\ell} S_2(\ell - \ell'')\right].
\end{aligned}$$

In this case, we have to add for each value of $-1 \leq r < \log_2(d^*) + 1$ with $d_{r-1} = 0$ and $\ell'' = \ell''(r)$:

$$H_3(n, r, \ell, \ell'') = f_1(r) 2^{-n} \left[2^{-1}\left(2^{\ell-\ell''} - 1\right)(n + \ell - \ell'') - 2^{\ell''-\ell} S_2(\ell - \ell'')\right]. \tag{13}$$

Case 4: $0 \leq p \leq 1$

If $p = 0$ or $p = 1$, there exists only one pair (m, c) which can produce the corresponding p. Thus, for each of these p's we have to add:

$$H_4(n, \ell) = 2^{-n-\ell}(n + \ell) \tag{14}$$

to the sum of the entropy.

The Algorithm 1 shows how we can compute the final entropy for an FCSR defined by n and d using the summands of the individual cases.

3.3 Complexity of the Computation

The exact computation of the entropy requires to evaluate the sums $S_1(k) = \sum_{x=1}^{2^k-1} x \log_2(x)$ and $S_2(k) = \sum_{x=2^{k-1}+1}^{2^k} x \log_2(x)$. If we have stored the values $S_1(k)$ and $S_1(k)$ for $1 \leq k \leq \ell$, we are able to compute the final entropy in $O(n^2)$. We need $O\left(2^\ell\right)$ steps to evaluate both sums, which is impractical for large ℓ. However, by using the bounds (20)-(21) for larger k's, we can easily compute a lower and upper bound of those sums. In Table 1, we compare for different

Algorithm 1. Final entropy

```
$H^f \leftarrow 0$
$\ell' \leftarrow 0$
$\ell \leftarrow wt(d) - 1$
$H^f \leftarrow H^f + 2H_4(n,\ell)$ $\{p = 0$ and $p = 1\}$
for $i = 1$ to $n - 1$ do
  if $d_{i-1} = 0$ then
    $H^f \leftarrow H^f + H_1(n,i,\ell,\ell')$
  else $\{d_{i-1} = 1\}$
    $\ell' \leftarrow \ell' + 1$
    $\ell'' \leftarrow 0$
    for $r = -1$ to $i - 1$ do
      if $d_{r-1} = 0$ then
        $H^f \leftarrow H^f + H_2(n,r,\ell,\ell',\ell'')$
      else $\{d_{r-1} = 1\}$
        $\ell'' \leftarrow \ell'' + 1$
      end if
    end for
  end if
end for
$\ell'' \leftarrow 0$
for $r = -1$ to $log_2(d - 2^{n-1})$ do $\{2^n \leq p \leq 2d - 1\}$
  if $d_{r-1} = 0$ then
    $H^f \leftarrow H^f + H_3(n,r,\ell,\ell'')$
  else $\{d_{r-1} = 1\}$
    $\ell'' \leftarrow \ell'' + 1$
  end if
end for
```

Table 1. Comparison of the exact computation of the final state entropy, with upper and lower bounds

n	d	ℓ	H^f	lb H^f	ub H^f	lb H^f , $k > 5$	ub H^f, $k > 5$
8	0xAE	4	8.3039849	8.283642	8.3146356	8.3039849	8.3039849
16	0xA45E	7	16.270332	16.237686	16.287598	16.270332	16.270332
24	0xA59B4E	12	24.273305	24.241851	24.289814	24.273304	24.273305
32	0xA54B7C5E	17		32.241192	32.289476	32.272834	32.272834

FCSRs the exact computation with those, using upper and lower bounds. The values of d were randomly chosen. However, the accuracy of the estimation of the sums can be shown anyway.

We mean by H^f the exact computation of the the final entropy. The values lb/ub H_f mean the lower and the upper bound obtained by using the approximation of S_1 and S_2. The last two columns, lb/ub H_f, $k > 5$, we gain by using the approximations only for $k > 5$. This last approximation is as close that we do not see any difference between the lower and the upper bound in the first 8 decimal places.

4 Lower Bound of the Entropy

We can use the algorithm of the previous section and induction to give a lower bound of the final state entropy. For a given n and ℓ we first compute the entropy for an FCSR where all the carry bits are the ℓ least significant bits. Subsequently, we show that by moving a feedback position to the left, the direction of the most significant bit, we increase the final entropy. In both steps, we study all $0 \leq p \leq |q|$ case by case and use the summands of the entropy H_1, H_2, H_3 and H_4 as presented in Section 3.

4.1 Basis of Induction

For a fixed n and ℓ we study the final state entropy of an FCSR with

$$d = 2^{n-1} + 2^{\ell} - 1.$$

This represents an FCSR which has all its recurrent positions grouped together at the least significant bits (see Fig. 5). Like in the previous section, we are going to compute the entropy case by case.

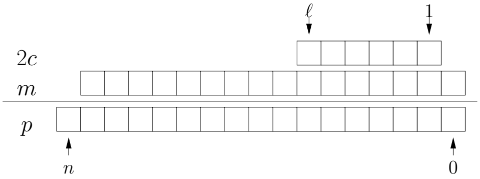


Fig. 5. FCSR with $d = 2^{\ell} - 1 + 2^{n-1}$

- $p = 0$ and $p = 1$.
- $1 \leq i \leq \ell$: Here we have $d_{i-1} = 1$, $\ell' = i$, $r = -1$ and 0 and thus $\ell'' = 0$.
- $\ell < i < n$: We have $d_{i-1} = 0$ and $\ell' = \ell$.
- $2^n \leq p \leq |q|$: Since it must hold that $r \leq \log_2(d^*) = \log_2(2^{\ell} - 1)$ we see that the only possible values for r are -1 and 0 and therefore $\ell'' = 0$.

So in this case, the final entropy is:

$$\begin{aligned} H^f(n,d) = \quad & 2H_4(n,\ell) \\ & + \sum_{i=1}^{\ell} \left(H_2(n,-1,\ell,i,0) + H_2(n,0,\ell,i,0)\right) \\ & + \sum_{i=\ell+1}^{n-1} H_1(n,i,\ell,\ell) \\ & + H_3(n,-1,\ell,0) + H_3(n,0,\ell,0) \\ = \quad & n + \ell\left(2^{-n+\ell+1} - 2^{-n+1}\right) - 2^{-n-\ell+2} S_2(\ell) \; . \end{aligned}$$

By using the lower bound (21) for $S_2(\ell)$ we can write:

$$H^f(n,d) \geq n + \frac{2^{-n+\ell+2}}{12\ln(2)}\left(3-(4+\ell)2^{-2\ell}-2^{-3\ell}+2^{1-4\ell}\right) .$$

Let us examine the function $g(\ell)=3-(4+\ell)2^{-2\ell}-2^{-3\ell}+2^{1-4\ell}$. It is easy to verify that $g(\ell+1)-g(\ell)>0$ for all $\ell \geq 1$. Thus, we can write $g(\ell) \geq g(1) = 7/4$ for all $\ell \geq 1$ and finally:

$$H^f(n,d) \geq n + 2^{-n+\ell}\frac{7}{12\ln(2)} \geq n . \tag{15}$$

4.2 Induction Step

We show that by moving one feedback position one position to the left, which means in direction of the most significant bit, the final state entropy increases. To prove this, we compare two cases $\mathcal{A}$ and $\mathcal{B}$. In Case $\mathcal{A}$ we choose an s such that $d_{s-1}=1$ and $d_s=0$. It must hold that:

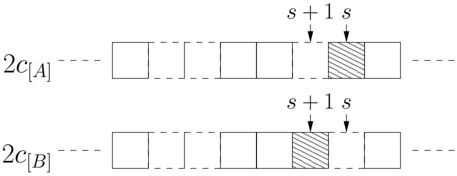


Fig. 6. Moving the feedback position

$$1 \leq s \leq n-2, \tag{16}$$
$$3 \leq n . \tag{17}$$

To create Case $\mathcal{B}$, we move the feedback at index s one position to the left. This is equivalent to:

$$d_{\mathcal{B}} = d_{\mathcal{A}} - 2^{s-1} + 2^s.$$

In Fig. 6, we display the two values $2c_{\mathcal{A}}$ and $2c_{\mathcal{B}}$ which we are going to use to compute the different finale entropies. Let

$$L = \#\{j < s | d_{j-1} = 1\}$$

be the number of feedback positions smaller than s. We mean by

$$I^r_{<s} = \{-1 \leq j < s : d_{i-1} = 0\}$$

the set of all indices smaller than s where there is no feedback and which are thus possible values for r. It must hold that $|I^r_{<s}| = s - L$.

We want to show that:

$$H^f_{\mathcal{B}} - H^f_{\mathcal{A}} \geq 0 \ . \tag{18}$$

To do this, we study the different cases of p. Each time, we examine if the summands from the algorithm in Section 3 are different or not. In the end we sum up the differences.

- $p = 0$ or $p = 1$: The summands of the entropy in $\mathcal{A}$ and $\mathcal{B}$ are the same.
- $i < s$: The summands of the entropy in $\mathcal{A}$ and $\mathcal{B}$ are the same.
- $n > i > s + 1$:
  - $d_{i-1} = 0$: In this case, i and ℓ' are the same for $\mathcal{A}$ and $\mathcal{B}$ and thus the summands of the entropy as well.
  - $d_{i-1} = 1$: We have:
    $$L + 2 \leq \ell' \leq \ell \ .$$
    - $r < s$: In this case, i, r, ℓ' and ℓ'' are the same and thus the summands of the entropy as well.
    - $s + 1 < r$: In this case, i, r, ℓ' and ℓ'' are the same and thus the summands of the entropy as well.
    - $s \leq r \leq s + 1$:
      - $\mathcal{A}$: Since for r it must hold that $d_{r-1} = 0$, we get $r = s + 1$ and thus $\ell'' = L + 1$. In this case, we have to count:
        $$H_2(n, s + 1, \ell, \ell', L + 1) \ .$$
      - $\mathcal{B}$: In this case, we have $r = s$ and $\ell'' = L$ and therefore the term:
        $$H_2(n, s, \ell, \ell', L) \ .$$
- $2^n \leq p \leq |q|$:
  - $r < s$: In this case, i, r and ℓ' are the same and thus the summands of the entropy as well.
  - $s + 1 < r$: In this case, i, r and ℓ' are the same and thus the summands of the entropy as well.
  - $s \leq r \leq s + 1$:
    - $\mathcal{A}$: We have $r = s + 1$ and $\ell'' = L + 1$. Therefore, the summand of the entropy in this case is:
      $$H_3(n, s + 1, \ell, L + 1) \ .$$
    - $\mathcal{B}$: We have to consider $r = s$ and $\ell'' = L$ and therefore:
      $$H_3(n, s, \ell, L) \ .$$
- $s \leq i \leq s + 1$: We are going to use $\ell''(r)$ to denote the value of ℓ'' corresponding to a specific r.
  - $i = s$:

- $\ast$ $\mathcal{A}$: In this case, we have $d_{i-1} = 1$ and $\ell' = L + 1$. Thus we have to count for each $r \in I^r_{<s}$:

$$H_2(n, r, \ell, L+1, \ell''(r)) \ .$$

- $\ast$ $\mathcal{B}$: Since $d_{i-1} = 0$ and $\ell' = L$ we get:

$$H_1(n, s, \ell, L) \ .$$

- $i = s + 1$:
  - $\ast$ $\mathcal{A}$: In this case, we have $\ell' = L + 1$ and $d_{i-1} = 0$, thus we need to consider:

$$H_1(n, s+1, \ell, L+1) \ .$$

  - $\ast$ $\mathcal{B}$: This time, we have $d_{i-1} = 1$. For $\ell' = L + 1$, $r \in I^r_{<s}$ and $r = s$ we get:

$$H_2(n, r, \ell, L+1, \ell''(r)) \ .$$

    In the case of $r = s$, we can write $\ell'' = L$.

By combining all these results, we get a difference of the final entropies of:

$$\begin{aligned}
H^f_{\mathcal{B}} - H^f_{\mathcal{A}} = & \sum_{\ell'=L+2}^{\ell} \left(H_2(n, s, \ell, \ell', L) - H_2(n, s+1, \ell, \ell', L+1)\right) \\
& + H_3(n, s, \ell, L) - H_3(n, s+1, \ell, L+1) \\
& + H_1(n, s, \ell, L) - \sum_{r \in I^r_{<s}} H_2(n, r, \ell, L+1, \ell''(r)) \\
& + \sum_{r \in I^r_{<s}} H_2(n, r, \ell, L+1, \ell''(r)) + H_2(n, s, \ell, L+1, L) \\
& - H_1(n, s+1, \ell, L+1) \\
& = 2^{\ell-1}\left(4\ell - 4L - 2\right) + 2^{2\ell-L} + 2^{L+1}\left(3S_2(\ell - L - 1) - S_1(\ell - L)\right) \ .
\end{aligned}$$

If we use the lower bound (21) for $S_2(\ell - L - 1)$ and the upper bound (20) for $S_1(\ell - L)$, we can write:

$$H^f_{\mathcal{B}} - H^f_{\mathcal{A}} \geq 2^L \left((\ell - L)\frac{1}{4\ln(2)} + \frac{7}{12\ln(2)} + 2^{-(\ell-L)}\frac{14 - 12\ 2^{-(\ell-L)}}{12\ln(2)} \right) \ .$$

From $\ell > L$, it follows directly (18), which means that the difference of the final entropies is greater or equal to 0.

Every FCSR with ℓ feedback positions can be build by starting with the FCSR described in Section 4.1 and successively moving one feedback position to the left. Thus, by combining (15) and (18) we write the following theorem.

Theorem 1. *An FCSR in Galois architecture, with given values for n and ℓ, has at least $n + 2^{\ell-n}\frac{7}{12\ln(2)} \geq n$ bits of entropy if all $2^{n+\ell}$ initial states appear with the same probability.*

5 Conclusion

If we allow all initial states of the FCSR with the same probability $2^{-n-\ell}$, we have an initial entropy of the state of $n+\ell$ bits. We showed in this article that already after one iteration, the entropy is reduced to $n+\ell/2$ bits.

As soon as the FCSR has reached its periodic behavior, the entropy does not reduce any more. In this article, we presented an algorithm which computes this final entropy in $O\left(n^2\right)$ steps. The algorithm is exact if the results of the sums $S_1(k)=\sum_{x=2^{k-1}+1}^{2^k} x\log_2(x)$ and $S_2(k)=\sum_{x=1}^{2^k-1} x\log_2(x)$ are known for $k\leq\ell$. For large values of ℓ, the same algorithm allows us to give close upper and lower bounds for the entropy by using approximations of $S_1(k)$ and $S_2(k)$.

In the end, we used the same algorithm to prove that the final state entropy never drops under n bits. One might argue that this is evident, since there are $|q|$ different values of p and $\log_2(|q|)\approx n$. However, it would be possible that the probabilities are not very regular distributed and that we would have a lower entropy. With our bound, it is sure that the entropy cannot drop under n bits.

The entropy of an FCSR decreases quite fast. However, it stays always larger or equal to n bits.

References

[AB05a] Arnault, F., Berger, T.P.: Design and properties of a new pseudorandom generator based on a filtered FCSR automaton. IEEE Transactions on Computers 54(11), 1374–1383 (2005)

[AB05b] Arnault, F., Berger, T.P.: F-FCSR: Design of a New Class of Stream Ciphers. In: Gilbert, H., Handschuh, H. (eds.) FSE 2005. LNCS, vol. 3557, pp. 83–97. Springer, Heidelberg (2005)

[ABL05] Arnault, F., Berger, T.P., Lauradoux, C.: F-FCSR. eSTREAM, ECRYPT Stream Cipher Project, Report 2005/008 (2005), http://www.ecrypt.eu.org/stream

[ABL06] Arnault, F., Berger, T.P., Lauradoux, C.: Update on F-FCSR stream cipher. In: SASC, State of the Art of Stream Ciphers Workshop, Leuven, Belgium, February 2006. ECRYPT Network of Excellence in Cryptology, pp. 267–277 (2006)

[ABM08] Arnault, F., Berger, T.P., Minier, M.: Some results on FCSR automata with applications to the security of FCSR-based pseudorandom generators. IEEE Transactions on Information Theory 54(2), 836–840 (2008)

[CL94] Couture, R., L'Ecuyer, P.: On the lattice structure of certain linear congruential sequences related to AWC/SWB generators. Math. Comput. 62(206), 799–808 (1994)

[GK02] Goresky, M., Klapper, A.: Fibonacci and galois representations of feedback-with-carry shift registers. IEEE Transactions on Information Theory 48(11), 2826–2836 (2002)

[KG93] Klapper, A., Goresky, M.: 2-adic shift registers. In: Anderson, R. (ed.) FSE 1993. LNCS, vol. 809, pp. 174–178. Springer, Heidelberg (1994)

[KG97] Klapper, A., Goresky, M.: Feedback shift registers, 2-adic span, and combiners with memory. J. Cryptology 10(2), 111–147 (1997)

[MZ91] Marsaglia, G., Zaman, A.: A new class of random number generators. Annals of Appl. Prob. 1(3), 462–480 (1991)

A Bounds for the Sums

In this section, we prove the following lower and upper bounds:

$$\sum_{x=2^{k-1}+1}^{2^k} x\log_2(x) \geq 2^{2k-3}\left(3k+1-\frac{3}{2\ln(2)}\right)+2^{k-2}(k+1)+\frac{1-2^{-k+1}}{24\ln(2)}\ , \quad (19)$$

$$\sum_{x=2^{k-1}+1}^{2^k} x\log_2(x) \leq 2^{2k-3}\left(3k+1-\frac{3}{2\ln(2)}\right)+2^{k-2}(k+1)+\frac{1-2^{-k}+3\,2^{1-2k}}{12\ln(2)},\quad (20)$$

$$\sum_{x=1}^{2^k-1} x\log_2(x) \geq 2^{2k-1}\left(k-\frac{1}{2\ln(2)}\right)-2^{k-1}k+\frac{4+k+2^{-k+1}}{24\ln(2)}\ , \quad (21)$$

$$\sum_{x=1}^{2^k-1} x\log_2(x) \leq 2^{2k-1}\left(k-\frac{1}{2\ln(2)}\right)-k\,2^{k-1}+\frac{4+k+2^{-k}-2^{1-2k}}{12\ln(2)}\ . \quad (22)$$

The idea of this proof is that:

$$\frac{1}{2}\left(x\log_2(x)+(x+1)\log_2(x+1)\right) \approx \int_x^{x+1} y\log_2(y)\,dy\ ,$$

since $\log_2(x)$ increases much slower than x and, thus, $x\log_2(x)$ is almost a straight line. This integral can be directly computed by:

$$\begin{aligned}\int_x^{x+1} y\log_2(y)\,dy &= \frac{y^2}{2}\left(\log_2(y)-\frac{1}{2\ln(2)}\right)\Bigg|_{y=x}^{x+1}\\ &= \frac{1}{2}\left(x\log_2(x)+(x+1)\log_2(x+1)\right)-\frac{1}{4}\frac{2x+1}{\ln(2)}\\ &\quad +\log_2\left(1+\frac{1}{x}\right)\frac{x}{2}(x+1)\ .\end{aligned}$$

We use the approximation of the natural logarithm:

$$\frac{1}{\ln(2)}\left(\frac{1}{x}-\frac{1}{2x^2}+\frac{1}{3x^3}-\frac{1}{4x^4}\right) \leq \log_2\left(1+\frac{1}{x}\right) \leq \frac{1}{\ln(2)}\left(\frac{1}{x}-\frac{1}{2x^2}+\frac{1}{3x^3}\right)$$

for $x > 0$ to get:

$$\frac{1+2x}{4\ln(2)}-\frac{\left(\frac{1}{3x}-\frac{1}{6x^2}+\frac{1}{2x^3}\right)}{4\ln(2)} \leq \frac{x}{2}(x+1)\log_2\left(1+\frac{1}{x}\right) \leq \frac{1+2x}{4\ln(2)}-\frac{\left(\frac{1}{3x}-\frac{2}{3x^2}\right)}{4\ln(2)}$$

and finally the bounds for the integral:

$$\int_x^{x+1} y\log_2(y)\,dy \geq \frac{1}{2}\left(x\log_2(x)+(x+1)\log_2(x+1)\right)-\frac{\left(\frac{1}{3x}-\frac{1}{6x^2}+\frac{1}{2x^3}\right)}{4\ln(2)} \quad (23)$$
$$\int_x^{x+1} y\log_2(y)\,dy \leq \frac{1}{2}\left(x\log_2(x)+(x+1)\log_2(x+1)\right)-\frac{\left(\frac{1}{3x}-\frac{2}{3x^2}\right)}{4\ln(2)}\ . \quad (24)$$

By combining the exact value of the integral:

$$\int_{2^{k-1}}^{2^k} y\log_2(y)\,dy = 2^{2k-3}\left(3k+1-\frac{3}{2\ln(2)}\right)$$

with the lower bound:

$$\begin{aligned}
&\int_{2^{k-1}}^{2^k} y\log_2(y)\,dy \\
&\geq \frac{1}{2}\sum_{x=2^{k-1}}^{2^k-1} x\log_2(x)+\frac{1}{2}\sum_{x=2^{k-1}+1}^{2^k} x\log_2(x)-\frac{1}{4\ln(2)}\sum_{x=2^{k-1}}^{2^k-1}\left(\frac{1}{3x}-\frac{1}{6x^2}+\frac{1}{2x^3}\right) \\
&= \sum_{x=2^{k-1}+1}^{2^k} x\log_2(x)-2^{k-2}(k+1)-\frac{1}{4\ln(2)}\sum_{x=2^{k-1}}^{2^k-1}\left(\frac{1}{3x}-\frac{1}{6x^2}+\frac{1}{2x^3}\right)\ .
\end{aligned}$$

gained by means of (23), we receive the upper bound:

$$\begin{aligned}\sum_{x=2^{k-1}+1}^{2^k} x\log_2(x) \leq 2^{2k-3}\left(3k+1-\tfrac{3}{2\ln(2)}\right)+2^{k-2}(k+1) \\ +\tfrac{1}{4\ln(2)}\textstyle\sum_{x=2^{k-1}}^{2^k-1}\left(\tfrac{1}{3x}-\tfrac{1}{6x^2}+\tfrac{1}{2x^3}\right)\ .\end{aligned} \quad (25)$$

In the same way, by using (24) we get:

$$\begin{aligned}\sum_{x=2^{k-1}+1}^{2^k} x\log_2(x) \geq 2^{2k-3}\left(3k+1-\tfrac{3}{2\ln(2)}\right)+2^{k-2}(k+1) \\ +\tfrac{1}{4\ln(2)}\textstyle\sum_{x=2^{k-1}}^{2^k-1}\left(\tfrac{1}{3x}-\tfrac{2}{3x^2}\right)\ .\end{aligned} \quad (26)$$

Let us have a closer look at the two functions $g_1(x)=\frac{1}{3x}-\frac{1}{6x^2}+\frac{1}{2x^3}$ and $g_2(x)=\frac{1}{3x}-\frac{2}{3x^2}$. If we analyze their first derivatives, we see that $g_1(x)$ is decreasing for $x\geq 1$ and $g_2(x)$ is decreasing for $x\geq 4$. For the upper bound of the sum, we can write directly:

$$\begin{aligned}
\sum_{x=2^{k-1}+1}^{2^k} x\log_2(x) &\leq 2^{2k-3}\left(3k+1-\frac{3}{2\ln(2)}\right)+2^{k-2}(k+1) \\
&\quad +\frac{1}{4\ln(2)}\sum_{x=2^{k-1}}^{2^k-1}\left(\frac{1}{3\ 2^{k-1}}-\frac{1}{6\ 2^{2k-2}}+\frac{1}{2\ 2^{3k-3}}\right) \\
&= 2^{2k-3}\left(3k+1-\frac{3}{2\ln(2)}\right)+2^{k-2}(k+1)+\frac{1-2^{-k}+3\ 2^{1-2k}}{12\ln(2)}
\end{aligned}$$

for all $k \geq 1$. In the case of the lower bound, we can write:

$$\sum_{x=2^{k-1}+1}^{2^k} x \log_2(x) \geq 2^{2k-3}\left(3k+1-\frac{3}{2\ln(2)}\right)+2^{k-2}(k+1)$$
$$+\frac{1}{4\ln(2)}\sum_{x=2^{k-1}}^{2^k-1}\left(\frac{1}{3\,2^k}-\frac{2}{3\,2^{2k-2}}\right)$$
$$=2^{2k-1}\left(k-\frac{1}{2\ln(2)}\right)-2^{k-1}k+\frac{4+k+2^{-k+1}}{24\ln(2)}$$

for $k \geq 3$. However, we can verify by numeric computation that the lower bound also holds in the cases $k = 1$ and $k = 2$. Thus, we have shown (19) and (20) for $k \geq 1$. Finally, by employing:

$$\sum_{x=1}^{2^K-1} x \log_2(x) = \sum_{k=1}^{K} \sum_{x=2^{k-1}+1}^{2^k} x \log_2(x) - K2^K$$

and the previous results, we receive the bounds (21) and (22).

B Proof of Lemma 1

Proof. Let p', m', c' be three bit strings of size k and let $X(p')$ be the number of possible pairs (m', c') such that $1p' = m' + c'$. We are going to use the following properties:

- For a given p', let (m', c') be such that $1p' = m'+c'$. We have two possibilities $10p' = 0m' + 1c' = 1m' + 0c'$ to create $10p'$. If (m', c') is such that $0p' = m' + c'$, we only have one possibility $10p' = 1m' + 1c'$ to build $10p'$. Thus, we can write:
$$X(0p') = 2\,X(p') + 1\,\left(2^k - X(p')\right). \tag{27}$$
- The only possibility to create $11p'$ is $11p' = 1m' + 1c'$ for a (m', c') such that $1p' = m' + c'$. Therefore, it holds that
$$X(1p') = X(p'). \tag{28}$$
- If the most significant bit of p' is a 0 followed by k times 1, we can only generate the carry in the last position. This is due to the fact that to create a carry and a 1 in the sum, we need three 1's which is not possible if we do not have a carry. So we have to create the carry in the highest position by $m'_k = c'_k = 1$. For the previous positions $0 \leq j < k$ we always have two possibilities for (m'_j, c'_j), respectively $(0, 1)$ and $(1, 0)$. Thus, in total we have 2^k choices for (m', c').
$$X(0\overbrace{1\ldots1}^{k}) = 2^k. \tag{29}$$

- As we have said above, it is not possible to create a carry with only 1's in p' and no previous carry, thus:

$$X(1\dots 1)=0. \tag{30}$$

We are now going to use induction over the length k of the bit strings p', m' and c' to show that for all $0 \le x \le 2^k-1$ there exists exactly one p' with $X(p')=x$.

Basis, $k=1$:
From (29) and (30) we can easily see that:

$$\begin{aligned} X(1)&=0,\\ X(0)&=1. \end{aligned}$$

Induction step, $k \rightarrow k+1$:
We assume that for every $0 \le x \le 2^k-1$ there exists a p' of length k such that $X(p')=x$. We want to show that the same assumption holds for $k+1$.

- $x=2^k$: From (29), we know that $X(p')=2^k$ if $p'=011\dots 1$ with k 1's.
- $0 \le x \le 2^k-1$: From the assumption, we know that there exists a p' of length k with $X(p')=x$, thus by using (28) we can write $X(1p')=x$.
- $2^k < x \le 2^{k+1}-1$: In this case $0 \le x-2^k \le 2^k-1$ and due to the assumption, we know that there exists a p' such that $X(p')=x-2^k$. By using (27) we get:

$$\begin{aligned} X(0p') &= 2X(p')+\big(2^k-X(p')\big)\\ &= 2(x-2^k)+(2^k-x+2^k)\\ &= x. \end{aligned}$$

We have proven that for all $0 \le x \le 2^k-1$ there exists a p' of length k, such that $X(p')=x$, *i.e.* there are exactly x pairs (m',c') of length k bits with $1p'=m'+c'$. Since in total there are only 2^k possible values of p' of length k, we see that there exists exactly one.

Bit-Pattern Based Integral Attack

Muhammad Reza Z'aba[1], Håvard Raddum[2,⋆], Matt Henricksen[3],
and Ed Dawson[1]

[1] Information Security Institute, Queensland University of Technology,
GPO Box 2434, Brisbane, Queensland 4001, Australia
m.zaba@isi.qut.edu.au, e.dawson@qut.edu.au
[2] Selmersenteret, University of Bergen, Norway
haavardr@ii.uib.no
[3] Institute for Infocomm Research, A*STAR,
21 Heng Mui Keng Terrace, Singapore 119613
mhenricksen@i2r.a-star.edu.sg

Abstract. Integral attacks are well-known to be effective against byte-based block ciphers. In this document, we outline how to launch integral attacks against bit-based block ciphers. This new type of integral attack traces the propagation of the plaintext structure at bit-level by incorporating bit-pattern based notations. The new notation gives the attacker more details about the properties of a structure of cipher blocks. The main difference from ordinary integral attacks is that we look at the pattern the bits in a specific position in the cipher block has through the structure. The bit-pattern based integral attack is applied to Noekeon, Serpent and PRESENT reduced up to 5, 6 and 7 rounds, respectively. This includes the first attacks on Noekeon and PRESENT using integral cryptanalysis. All attacks manage to recover the full subkey of the final round.

Keywords: Block ciphers, integral cryptanalysis, Serpent, Noekeon, PRESENT.

1 Introduction

The integral attack [11] is the basis for the best attacks on the AES, and has become standard in a cryptanalyst's toolbox. The basic idea of the attack is to analyze how a specified property of a set of plaintexts will evolve through the encryption algorithm and to use the existence of that property to verify key guesses. Up until now, integral attacks have not been thought suitable for bit-based ciphers. In these attacks the plaintext bytes are chosen to be constant, or take on all values through the set of texts. The reason for this choice is that it is unaffected by the application of a bijective S-box substituting bytes.

Using the traditional approach on a bit-oriented cipher, the bits output from an S-box are not treated as a block. This normally implies that any all-values property of the S-box output will be subsequently destroyed by the linear layer.

[⋆] This work done while visiting the Information Security Institute, Queensland University of Technology, Australia.

K. Nyberg (Ed.): FSE 2008, LNCS 5086, pp. 363–381, 2008.

In order to address this issue, we introduce a new bit-pattern based notation. Each bit position within a structure holds a specific sequence of bit '0' and '1'. The pattern in which the bit sequence is repeated serves as the basis of the notation. This means that the *order* of the texts in a bit-pattern based integral attack plays an important part, in contrast to the usual integral attack where the texts are regarded as an unordered set. This allows an attacker to gain knowledge of bit patterns in the set of texts through some encryption rounds. Instead of inputting all possible values into a single S-box in the first round, the bit-pattern based structure is constructed such that the active bits are spread over more than one S-box.

The bit-pattern based integral attack manage to penetrate up to 5 (out of 16), 6 (out of 32) and 7 (out of 31) rounds of Noekeon [9], Serpent [1] and PRESENT [7], respectively. To the best of our knowledge, this is the first integral cryptanalysis on Noekeon and PRESENT reported in the literature. For all three ciphers, detailed analysis by the designers show the minimal complexity for a successful differential attack on a few rounds. Bit-pattern based integral cryptanalysis gives much better results for these reduced-round ciphers. Note, however, that differential cryptanalysis can be easily extended to more rounds whereas integral cryptanalysis can not be extended beyond a certain point.

This document is organized as follows. The integral attack and the new notations are explained in Section 2. The attacks on Noekeon, Serpent and PRESENT are presented in Section 3. Section 4 highlights some related work. Discussions and conclusion are given in Section 5.

2 Bit-Pattern Based Integral Attack

An integral attack works by choosing a set of plaintexts, where some bit positions take on all values through the set, and the other bits are chosen to be arbitrary constants. The set of cipher blocks used in an integral attack is commonly referred to as a *structure*. The part of the structure that takes on all values usually forms the input to one or more bijective S-boxes in the first round. Then two properties are achieved by the structure: it is unaffected by key-addition, and it is unaffected by the application of bijective S-boxes. The diffusion of the cipher will however mix the bits, so after some time the inputs of some S-boxes will not have the constant or all-values property. When the cipher is byte-oriented (like the AES), the bits in the output of an S-box are treated as a block, and only mixed with blocks of bits that have either the all-values or the constant property. This might delay the destruction of these properties in the structure, and lead to good attacks.

For a bit-based cipher, the bits in the output of one S-box are treated independently and are not mixed in a way that respects the S-box boundaries. Hence the input bits to an S-box in the next round will have some constant bits, and some bits that are not constant. The useful properties of a structure will then be lost already in the second round. We overcome this and make integral attacks possible on bit-based cipher by introducing a more refined notation for the bits in a structure.

2.1 Pattern-Based Notations

In our bit-pattern based approach, the status of each single bit position within the overall structure is treated independently. Each bit position in a plaintext structure holds a specific sequence of bit '0' and/or '1'. The pattern in which the bit sequence is repeated forms the foundation of the bit-based notations. The following describes the notation.

- The pattern c in a position means that all bits in this position within the structure consists of only bit '0' or '1'. This pattern is called a **constant** bit pattern.
- The pattern a_i in a position means that the first block of 2^i consecutive bits in this position are constant, and the next block of 2^i consecutive bits all have the opposite value of the first. The alternating values of bits in 2^i-blocks is repeated throughout the structure. This pattern is called an **active** bit pattern.
- The pattern b_i in a position means that blocks of 2^i consecutive bits in this position are constant, but the values of the blocks are not necessarily repeated in an alternating manner.
- The pattern d_i in a position means that the bits in this position may hold either a c (constant) or an a_i (active) pattern. This pattern is called a **dual** bit pattern.

If the XOR sum of all the bits in one pattern equals 0, we say that the pattern is *balanced*. Furthermore, if the cipher block which is the XOR sum of all the texts in a structure only has 0-bits we say that the structure is balanced. All patterns described above are balanced, except for the b_0-pattern which may or may not be balanced. In fact, any bit-string fulfills the definition of a b_0-pattern. To make a distinction between balanced and unbalanced b_0-patterns, we will write b_0^* when we know that the pattern is balanced and b_0 otherwise.

As an example, possible values of a 4-bit text structure with the patterns $\mathsf{a}_0\mathsf{a}_3\mathsf{c}\mathsf{a}_2$ are $\{6_x, E_x, 6_x, E_x, 7_x, F_x, 7_x, F_x, 2_x, A_x, 2_x, A_x, 3_x, B_x, 3_x, B_x\}$. Table 4 in the Appendix lists out the possible values of c, a_i and some b_i patterns in a structure of 2^4 texts.

2.2 Tracing Bit Patterns through the Cipher

Bit-patterns will be XORed together in the linear operations of the cipher. The following properties are easy to verify.

- $\mathsf{c} \oplus \mathsf{p} = \mathsf{p}$ for any pattern p.
- $\mathsf{a}_i \oplus \mathsf{a}_i = \mathsf{c}$.
- $\mathsf{p}_j \oplus \mathsf{q}_i = \mathsf{b}_i$ for $j > i$ and $\mathsf{p}, \mathsf{q} \in \{\mathsf{a}, \mathsf{b}\}$. If $i = 0$ the right-hand side will be b_0^*.
- $\mathsf{p} \oplus \mathsf{b}_0^* = \mathsf{b}_0^*$ when $\mathsf{p} \neq \mathsf{b}_0$.

These rules of XOR addition will be used when analyzing how the bit-patterns in the cipher block evolves through the linear parts of a cipher.

When the bit-patterns pass through an S-box, every output-bit of the S-box will have a b_i-pattern where i is the smallest index found in the input patterns. This is because blocks of 2^i inputs will all have the same value, and so the output values will also appear in blocks of 2^i equal values.

That is all that can be said in general when patterns pass through an S-box, but there is another fact that can be useful when analyzing the effect an S-box has on input patterns. It is summed up in the following lemma.

Lemma 1. *Consider m bit sequences, expressed as linear combinations of a_i-patterns $l_1, \ldots, l_m$, where $i \leq n$. Write this using matrix notation as $M\mathsf{a} = \mathbf{l}$, where $\mathsf{a} = (\mathsf{a}_0, \ldots, \mathsf{a}_n)^T$ and $\mathbf{l} = (l_1, \ldots, l_m)^T$. The different values for the m bits found in the same position in the sequences lie in an affine space of size $2^{rank(M)}$.*

Proof: Let $r = m - \text{rank}(M)$. Then there exists r linearly independent vectors $\mathbf{v}_1, \ldots, \mathbf{v}_r$ such that $v_i M = \mathbf{0}$, $i = 1, \ldots, r$. Since $\mathsf{a}_i \oplus \mathsf{a}_i = \mathsf{c}$ in our context, a 0-row in M corresponds to the constant pattern. This means that all possible values of the m bits lie in an affine space cut out by the r linear equations given by $\mathbf{v}_1, \ldots, \mathbf{v}_r$ and r right-hand sides. The size of this space is 2^{m-r} and the lemma follows. □

The lemma above can be helpful when determining whether the balancedness of a structure is lost through the application of an S-box. Assume we have an m-bit S-box and a structure of 2^n texts where $m > n$, and assume the input bits to the S-box are expressed as linear combinations of a_i-patterns. Suppose Lemma 1 tells us the inputs to the S-box lie in an affine space of dimension smaller than n. Then each distinct input value will occur an even number of times, and so each distinct output value will occur an even number of times. Hence the balancedness will not be lost after the S-box.

2.3 Generic Bit-Pattern Based Integral Attack

Here we describe a generic bit-pattern based integral attack that can be used on Noekeon, Serpent and PRESENT. These ciphers are similar in structure, so we will use the same notation on all of them.

The input to round i is denoted by $X_i = (x_0^i, x_1^i, x_2^i, x_3^i)$ where X_0 is the plaintext. The input and output of the S-box layer in round i are denoted by $Y_i = (y_0^i, y_1^i, y_2^i, y_3^i)$ and $Z_i = (z_0^i, z_1^i, z_2^i, z_3^i)$, respectively. The round i subkey is denoted by $K_i = (k_0^i, k_1^i, k_2^i, k_3^i)$. The blocks X_i, Y_i, Z_i and K_i consist of four 32-bit words for Noekeon and Serpent, and four 16-bit words for PRESENT. In every word, the rightmost bit is Bit '0' and $x_{[\ell]}$ denotes the ℓ-th bit of x. All non-linear components in these ciphers are composed of 4×4 bijective S-boxes.

The attacker first finds a structure of plaintexts, and sees how the bit-patterns of the structure become affected through the cipher. Just before the S-box layer in some round the structure will be balanced, but the balancedness is expected to be destroyed after the S-box layer. If this happens in round r, the following equation must hold:

$$\bigoplus_{j=0}^{m-1} Y_r^{(j)} = \bigoplus_{j=0}^{m-1} S^{-1}(Z_r^{(j)}) = 0 \tag{1}$$

where m is the size of the structure. We then guess enough key material so we can partially decrypt the ciphertexts to find all the bits coming out of one of the S-boxes in round r, and use Equation (1) to verify the guess.

Equation (1) puts a 4-bit condition on the guess, so we expect the number of possible key-bit guesses to be reduced by a factor 2^{-4}. If we are guessing on k key-bits at the same time, we will then need approximately $\lceil k/4 \rceil$ structures to identify the correct parts of the round keys used in the last rounds.

This can be summed up in Algorithm (1), where we assume we need to guess k bits from the last round key(s).

```
Precomputation
Analyze round function to identify distinguisher;
begin
    Choose a structure of plaintexts that matches distinguisher;
    Encrypt all plaintexts in structure and get corresponding ciphertexts;
    Initialize an array $A[]$ of size $2^k$ bits with all ‘1’s;
    Set $v = 0$;
    while number of entries such that $A[v] = 1$ is greater than one do
        Partially decrypt all ciphertexts using the value $v$ as partial subkey bits
        to find the output bits of one S-box in round $r$;
        if Equation (1) does not hold then
            set $A[v] = 0$;
        end
        $v = v + 1$;
    end
    Output value $v$ for which $A[v] = 1$ as correct subkey bits;
end
```

Algorithm 1. Algorithm for basic attack

We may also extend an attack by one round by adding one round in the beginning. This can be done by letting the bits in the structure have a specific pattern at the input of the second round, instead of in the plaintexts. These patterns are then traced backwards through the first round, until they meet the output of the S-box layer in the first round. S-boxes that have a sum of active patterns in its output bits are called active S-boxes. By specifying a value of the starting bit for patterns in the output of the active S-boxes, we specify some values of these outputs, and can find the values of the inputs. Next we guess the value of the bits in the key used for pre-whitening that affect the active S-boxes in the structure. This allows us to find the structure of plaintexts that will have the specific bit-pattern at the input of the second round, when the guess of bits in the pre-whitening key is right.

If we need to guess k bits from the pre-whitening key, we must expect to use 2^k structures before we get one with the specified patterns in the second round. This increases the number of chosen plaintexts needed, but it may involve a smaller guess on key-material than would be needed by adding a round at the end.

3 Application on Noekeon, Serpent and PRESENT

We have used bit-pattern based integral cryptanalysis on the block ciphers Noekeon, Serpent and PRESENT. Here we show how the attacks worked.

3.1 Noekeon

Noekeon [9] accepts a 128-bit block of plaintext X_0 and a 128-bit key. The 128-bit block of ciphertext X_{17} is produced after iterating a round function 16 times, followed by a final output function. The round function consists of two linear layers, L_0 and L_1, and one non-linear layer S. The final round involves only L_0. The encryption scheme of Noekeon can be depicted as:

$$\begin{aligned} X_{i+1} &= L_1^{-1}(S(L_1(L_0(X_i, K)))), i = 0, 1, \ldots 15 \\ X_{17} &= L_0(X_{16}, K) \end{aligned}$$

Figure 1 illustrates the round function of Noekeon. The same round subkey is used in every round. Let $x \lll i$ and $x \ggg i$ imply the rotation of the word x by i bits to the left and right, respectively. The linear layer L_0 of Noekeon is described as:

$$t_0^i = (x_0^i \oplus c^i \oplus k_0 \oplus u^i) \tag{2}$$

$$t_1^i = (x_1^i \oplus k_1 \oplus v^i) \tag{3}$$

$$t_2^i = (x_2^i \oplus k_2 \oplus u^i) \tag{4}$$

$$t_3^i = (x_3^i \oplus k_3 \oplus v^i) \tag{5}$$

where $u^i = R(p^i)$, $v^i = R(q^i)$, $p^i = x_1^i \oplus k_1 \oplus x_3^i \oplus k_3$, $q^i = x_0^i \oplus c^i \oplus x_2^i$, $R(x) = x \oplus (x \lll 8) \oplus (x \ggg 8)$ and c^i is a round constant. L_1 simply consist of three rotations of the words in the cipher block $(y_0^i, y_1^i, y_2^i, y_3^i) = (t_0^i, t_1^i \lll 1, t_2^i \lll 5, t_3^i \lll 2)$.

3.5-Round Distinguisher. Prepare a structure of 2^{16} plaintexts:

$$X_0^{(j)} = (x_0^{0(j)}, x_1^{0(j)}, x_2^{0(j)}, x_3^{0(j)}) = (j \| c_0, R(j \| c_1), c_2, R(j \| c_3))$$

where $0 \leq j \leq 2^{16} - 1$, c_0, c_1, c_3 are arbitrary 16-bit constants and c_2 is a 32-bit constant. By consulting Equations (2),(3),(4) and (5), it can be observed that u^i will become a constant and v^i will cancel the active bits in x_1^0 and x_3^0. This leaves the 16 leftmost bits of y_0^i to hold active patterns, i.e. $\mathsf{a}_{15}\mathsf{a}_{14} \ldots \mathsf{a}_0$. All other bits hold c patterns. The propagation of bit patterns in this distinguisher is shown in Figure 4 in the appendix.

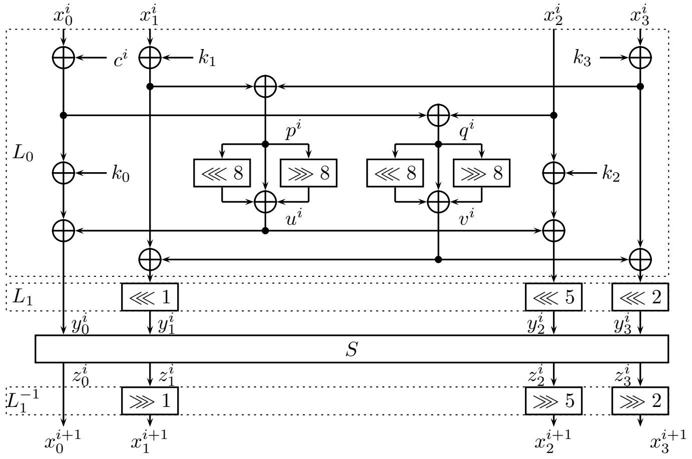


Fig. 1. Round function of Noekeon in Round i

There are 16 active S-boxes at the input of S in the first round. The remaining 16 S-boxes receive an all c input patterns. Each active S-box has two inputs which differ only in the leftmost bit. There exists a partial differential through the S-box $8_x \rightarrow w \| 3_x$ with probability 1, where $w \in \{0_x, 1_x, 2_x, 3_x\}$. As a consequence, the 16 leftmost bits of both $z_2^{0(j)}$ and $z_3^{0(j)}$ assume the same a_i pattern as the leftmost bit of the input. The rest of the output bits of the active S-boxes hold a d_i pattern where $0 \leq i \leq 15$.

In the second round, the linear combinations of c and a_i bits inside L_0 guarantee that no c pattern remains in any bit position. Note that the partial differential plays a critical role to ensure that this property occurs with certainty. Every bit of u^1 and v^1 contains at least one a_i pattern. L_1 ensures that all bit patterns in every column are linearly independent. According to Lemma 1 there are therefore 16 distinct values in the input and output of every S-box, which are repeated 2^{12} times. After the linear operations in the second round, the number of times each distinct value appears in the input of any S-box is still even, so the bits assume a b_0^* pattern after the S-box layer.

Experimentally, it has been verified that L_0 and L_1 in the third round do not cause any value in any input to an S-box to occur an odd number of times. At the input of S, the number of different inputs into each S-box is even and therefore, the number of different outputs is also even. This causes the structure to remain balanced after S.

In the fourth round, the balancedness of the structure is ensured through L_0 and L_1, but is expected to be destroyed after the application of S.

Key Recovery. The 3.5-round distinguisher can be used to attack four and five rounds of Noekeon using the attack strategy described in Section 2.3.

The key recovery procedure in a 4-round attack is a straightforward process. Once the distinguisher is available, Equation (1) must hold for $m = 2^{16}$ and $r = 3$.

The following equations provide the output bits of the S-boxes in the fourth round:

$$z^{3(j)}_{0[\ell]} = (x^{4(j)}_0 \oplus R(x^{4(j)}_1 \oplus x^{4(j)}_3) \oplus c^4)_{[\ell]} \oplus k_{0[\ell]} \tag{6}$$

$$z^{3(j)}_{1[\ell]} = (x^{4(j)}_1 \oplus R(x^{4(j)}_0 \oplus x^{4(j)}_2))_{[\ell-1]} \oplus A_{1[\ell-1]} \tag{7}$$

$$z^{3(j)}_{2[\ell]} = (x^{4(j)}_2 \oplus R(x^{4(j)}_1 \oplus x^{4(j)}_3))_{[\ell-5]} \oplus k_{2[\ell-5]} \tag{8}$$

$$z^{3(j)}_{3[\ell]} = (x^{4(j)}_3 \oplus R(x^{4(j)}_0 \oplus x^{4(j)}_2))_{[\ell-2]} \oplus A_{3[\ell-2]} \tag{9}$$

where $[\ell + n]$ is computed modulo 32 and $A_{1[\ell]}$ and $A_{3[\ell]}$ are linear combinations of seven key bits as follows:

$$\begin{aligned} A_{i[\ell]} &= (R(k_0 \oplus k_2) \oplus k_i)_{[\ell]} \\ &= (k_0 \oplus k_2)_{[\ell]} \oplus (k_0 \oplus k_2)_{[\ell+8]} \oplus (k_0 \oplus k_2)_{[\ell-8]} \oplus k_{i[\ell]}. \end{aligned} \tag{10}$$

For the 4-round attack, we need to guess on 4 bits of key material at the same time; the bits $k_{0[\ell]}$ and $k_{2[\ell-5]}$ and the values of the linear combinations $A_{1[\ell-1]}$ and $A_{3[\ell-2]}$. This means we should need approximately one structure to identify a correct guess, in practice we sometimes need 2. This needs to be repeated 32 times to get 128 bits of key material from the last round key. After the correct values for $k_{0[\ell]}$, $k_{2[\ell]}$, $A_{1[\ell]}$ and $A_{3[\ell]}$ are identified for $\ell = 0, 1, \ldots, 31$, Equation (10) is rearranged and solved to uncover the unknown bits in k_1 and k_3. The attack requirements are $2 \times 2^{16} = 2^{17}$ chosen plaintexts and $2 \times 2^{16} \times 2^4 \times 32 = 2^{26}$ partial decryptions.

In a 5-round attack, the values of the outputs from one S-box in the third round can be obtained by guessing 92 selected bits of information from the keys used in round 5 and 4. Here we make use of the fact that Noekeon uses the same key in every round, so the key material we need to guess in round 4 overlaps with what we need to guess in round 5.

In order to correctly identify all the 92 bits, we have to use 23 different structures. The remaining $128 - 92 = 36$ bits can be found by exhaustive search. The number of plaintexts required is therefore $23 \times 2^{16} \approx 2^{20.6}$ chosen plaintexts. The time complexity for the attack is $(2^{92} + 2^{88} + \ldots + 2^4 + 1) \times 2^{16} + 2^{36} \approx 2^{108.1}$ partial decryptions. Memory is required for storing 2^{92} bits indicating possible guesses remaining, thus the memory requirement is 2^{89} bytes.

3.2 Serpent

Serpent [1] is a 128-bit block cipher with key sizes between 0 to 256 bits. It has 32 rounds and can be represented in non-bit-sliced and bit-sliced version. We focus on the bit-sliced version of Serpent. The round function is composed of a key mixing layer, a non-linear S-box layer S_i, and a linear transformation layer L. In the last round, the linear transformation is replaced with a key mixing layer. The round function of Serpent in Round i is depicted in Figure 2. The cipher can be expressed by the following equations:

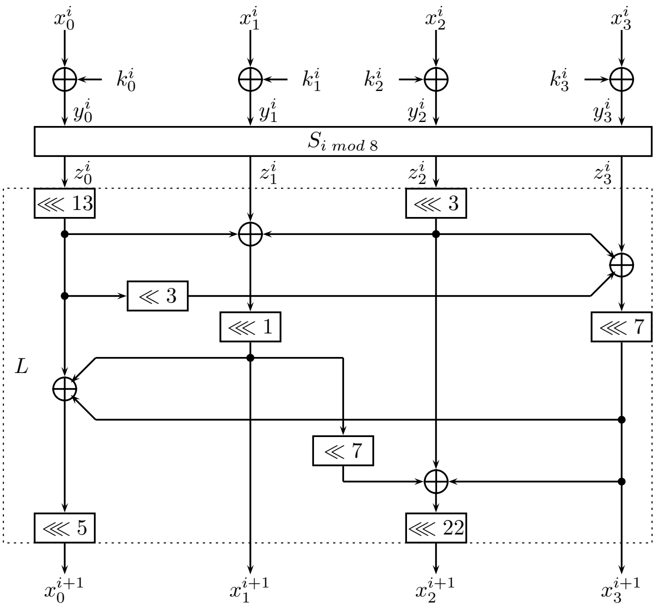


Fig. 2. Round function of Serpent in Round i

$$\begin{aligned} X_{i+1} &= L(S_{i\ mod\ 8}(X_i \oplus K_i)), i = 0, 1, \ldots 30 \\ X_{32} &= S_7(X_{31} \oplus K_{31}) \oplus K_{32}. \end{aligned}$$

Serpent has eight different 4 × 4 S-boxes. Round i uses S-box $S_{i\ mod\ 8}$ 32 times in parallel. The input is taken one bit from each 32-bit word of the same bit position. The linear transformation of Serpent can be expressed as:

$$\begin{aligned} x_{0[\ell]}^{i+1} &= z_{0[\ell-18]}^i \oplus z_{1[\ell-6]}^i \oplus z_{0[\ell-19]}^i \oplus z_{2[\ell-9]}^i \oplus \\ &\quad z_{3[\ell-12]}^i \oplus z_{2[\ell-15]}^i \oplus (z_{0[((\ell-12)\ll 3)-13]}^i) \\ x_{1[\ell]}^{i+1} &= z_{1[\ell-1]}^i \oplus z_{0[\ell-14]}^i \oplus z_{2[\ell-4]}^i \\ x_{2[\ell]}^{i+1} &= z_{2[\ell-25]}^i \oplus z_{3[\ell-29]}^i \oplus z_{2[\ell]}^i \oplus (z_{0[((\ell-29)\ll 3)-13]}^i) \oplus \\ &\quad z_{1[((\ell-22)\ll 7)-1]}^i \oplus z_{0[((\ell-22)\ll 7)-14]}^i \oplus z_{2[((\ell-22)\ll 7)-4]}^i \\ x_{3[\ell]}^{i+1} &= z_{3[\ell-7]}^i \oplus z_{2[\ell-10]}^i \oplus z_{0[((\ell-7)\ll 3)-13]}^i \end{aligned}$$

where $[\ell \ll n] = [\ell - n]$ is not computed modulo 32. If $(\ell \ll n) < 0$ then the bit at the position is a zero bit.

The cipher has endured extensive cryptanalysis [12,10,2,3,4,5,8] but no analysis on integral attack has been reported.

3.5-Round Distinguisher. Serpent reduced to 3.5 rounds can be distinguished from a random permutation by choosing a structure of 2^{10} plaintexts. The plaintexts are chosen such that the ten most significant bits of x_2^0 hold all possible 10-bit values. The rest of the plaintext bits hold constant values such as the following:

$$X_0^{(j)} = (x_0^{0(j)}, x_1^{0(j)}, x_2^{0(j)}, x_3^{0(j)}) = (c_0, c_1, j\|c_2, c_3)$$

where $0 \leq j \leq 2^{10} - 1$, c_0, c_1, c_3 are 32-bit constants and c_2 is a 22-bit arbitrary constant. The ten leftmost bits of $x_2^{0(j)}$ therefore hold the pattern $\mathsf{a}_9\mathsf{a}_8 \ldots \mathsf{a}_0$ and the rest of the bits hold a c pattern. The bit pattern propagation of the 4-round distinguisher is shown in Figure 5 in the appendix.

In the first round, the inputs to the ten affected instances of the S-box receive a pair of inputs with a difference of 4_x. Each input value is repeated 2^9 times. These S-boxes will, therefore, output a pair of values repeated 2^9 times. Since each of the eight S-boxes of Serpent behaves differently to the input difference, the output bits of these S-boxes are denoted as a d_i-pattern. However, at least two bits in each output will hold an a_i-pattern. This is due to one of the design criteria of the S-box, i.e., one-bit input change results in at least two-bit output change. All other bits remain constant. The linear layer in the first round does not affect the balancedness of the structure.

Each instance of the second round S-box may receive a single constant input value, or between two to sixteen different input values. The input values are repeated between 2^6 and 2^{10} times. The number of distinct outputs is therefore even and this ensures that the structure is balanced after the S-box in the second round.

The linear layer in the second round ensures that the number of distinct values in every column occurs an even number times, or that all values occur an odd number of times. All bits will hold a b_0^*-pattern except for a few positions which still retain a b_1-pattern. These bits are then fed to the third round S-box. The number of repetition for each distinct output value matches that of its input and this preserves the balancedness of the structure until after L in the third round.

The application of the fourth round S-boxes is expected to destroy the balancedness of the structure.

Key Recovery. Note that in the last round L is replaced with key addition, so for four-round Serpent the ciphertexts and the output of the S-boxes in the fourth round is only separated by a simple XOR of the last round key. Hence we only need to guess four bits of the last round key to make use of Equation (1). This attack requires 2×2^{10} chosen plaintexts, and it must be repeated 32 times to recover the whole last round key. The time complexity is therefore $2 \times 2^{10} \times 2^4 \times 32 = 2^{20}$ partial decryptions.

In a 5-round attack, we verify Equation (1) three times, for the fourth round S-boxes in bit positions 0,1 and 2. To compute the output of the S-box in position 0, we need to guess 11 bits of K_4 and 36 bits of K_5. For position 1 we need to guess 32 bits of K_5 and 10 bits of K_4, and for position 2 we need to guess 20 bits of K_5 and 10 more bits of K_4. The remaining 40 bits of K_5 can then be found by exhaustive search. In order to correctly identify all bits 12 structures are needed, so a total of $12 \times 2^{10} \approx 2^{13.6}$ chosen plaintexts are needed. The time requirement is approximately $(2^{47} + 2^{43} + \ldots + 1) \times 2^{10} \times 3 + 2^{40} \approx 2^{58.7}$ partial decryptions. The memory for the possible key candidates are 2^{44} bytes.

Table 1. Bit patterns of Z_0 in the 6-round attack

Word	Bit Pattern
z_0^0	c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c c
z_1^0	c c c c c c c c c c c c c c c c c c c c c c a_9 a_8 a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0
z_2^0	a_2 a_1 a_0 c c c c c c c c c c c c c c c c c c c c c c a_9 a_8 a_7 a_6 a_5 a_4 a_3
z_3^0	c c c c c c c c c c c c c c c c c c c c c c a_9 a_8 a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0

The 5-round attack can be extended by adding one round at the beginning. We want the 10 most significant bits of x_2^1 to hold the pattern $\mathsf{a}_9\mathsf{a}_8\ldots\mathsf{a}_0$ and the rest of the bits to hold a c pattern. If the active bits in x_2^1 are traced backwards until the input of the linear transformation of the first round (Round 0), the bits assume the pattern shown in Table 1. Since the number of active S-boxes is 13, we need to guess 52 bits of K_0 to find a plaintext structure that will evolve into the desired pattern in X^1 when the guess is right. This requires $2^{13.2} \times 2^{52} \approx 2^{65.2}$ chosen plaintexts and approximately $2^{58.7} \times 2^{52} \approx 2^{110.7}$ partial decryptions. The memory requirement is the same as in the 5-round attack.

3.3 PRESENT

PRESENT [7] is a 64-bit block cipher with key sizes of 80 and 128 bits. It has 31 rounds and is developed exclusively for lightweight applications. Figure 3 illustrates the round function of this cipher. The encryption function is given as:

$$X_{i+1} = L(S(X_i \oplus K_i)), i = 0, 1, \ldots 30$$
$$X_{32} = X_{31} \oplus K_{31}$$

For consistency with the other analyzed ciphers, the representation of the bits are slightly modified so that the 16 S-boxes in S are implemented in bit-sliced mode like Noekeon and Serpent. The linear layer L is described in Table 2.

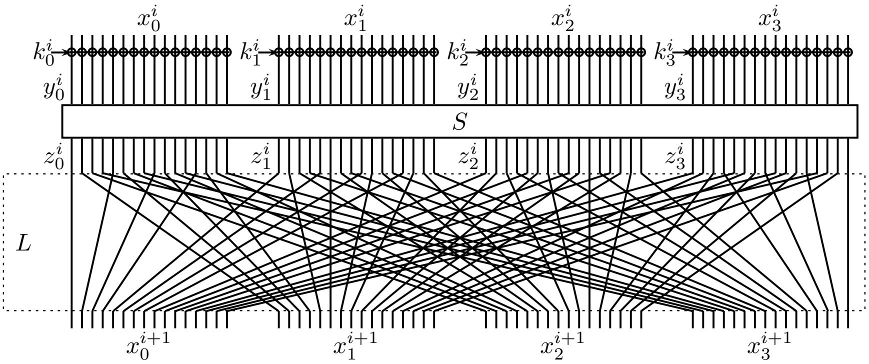


Fig. 3. Round function of PRESENT in Round i

Table 2. Linear layer L of PRESENT

Output	Input			
	z_0^i	z_1^i	z_2^i	z_3^i
x_0^{i+1}	15 11 7 3	15 11 7 3	15 11 7 3	15 11 7 3
x_1^{i+1}	14 10 6 2	14 10 6 2	14 10 6 2	14 10 6 2
x_2^{i+1}	13 9 5 1	13 9 5 1	13 9 5 1	13 9 5 1
x_3^{i+1}	12 8 4 0	12 8 4 0	12 8 4 0	12 8 4 0

3.5-Round Distinguisher. A 3.5-round distinguisher can be built for PRESENT by constructing a structure of 2^4 chosen plaintexts:

$$(x_0^{0(j)}, x_1^{0(j)}, x_2^{0(j)}, x_3^{0(j)}) = (c_0, c_1, c_2, c_3 \| j)$$

where c_0, c_1, c_2 are arbitrary 16-bit constants, c_3 are random 12-bit constants and $0 \leq j \leq 15$. The bit propagation of this distinguisher is shown in Figure 6 in the appendix.

In the first round, each of the four rightmost S-boxes receives two different input values repeated eight times. The rest of the S-boxes receives only a single constant value repeated sixteen times. The output bits of the four rightmost S-boxes assume the pattern $\mathsf{d}_i\mathsf{d}_i\mathsf{d}_i\mathsf{a}_i$ since there is a differential $1_x \rightarrow w \| 1_x$ occurring with probability 1 where $w \in \{1_x, 3_x, 4_x, 6_x\}$.

In the second round, these sixteen bit patterns are fed to S-boxes 0, 4, 8 and 12. S-box 0 receives the pattern $\mathsf{a}_3\mathsf{a}_2\mathsf{a}_1\mathsf{a}_0$ which represents all possible 4-bit values. S-boxes 4, 8 and 12 receive the pattern $\mathsf{d}_3\mathsf{d}_2\mathsf{d}_1\mathsf{d}_0$. The inputs to the other 12 S-boxes have the pattern c. The input and output values of the active S-boxes are repeated either once or an even number of times, the structure therefore is balanced.

The linear layer in the second round spreads the bits such that each S-box in the third round has the pattern cccb_0^*. Since only one bit position is non-constant, all S-boxes receive at most two different input values. In the preceding round, the output of S-box 0 consists of all possible 4-bit values. Therefore, due to the linear transformation, the number of repetitions for the different input values for S-boxes 0, 4, 8 and 12 in the current round is exactly 8. The output bits of all S-boxes at this point hold the pattern b_0^* and the structure remains balanced.

In the fourth round, the balancedness of the structure is expected to be destroyed after the application of the S-box.

Key Recovery. The attack on 4 rounds is exactly the same as for Serpent. The number of chosen plaintexts needed is $2 \times 2^4 = 2^5$ with time complexity of $2 \times 2^4 \times 16 \times 2^4 = 2^{13}$ partial decryptions.

In a 5-round attack, due to the linear layer, the attacker needs to guess an additional $4 \times 4 = 16$ bits of key material from K_5, so 5 structures are needed to identify the correct guess. The attack can be repeated 3 times to get 60 bits of K_5, the remaining 20 bits of K_5 can be found by exhaustive search. The number of chosen plaintexts needed is $5 \times 2^4 \approx 2^{6.4}$, and the time complexity is $(2^{20} + 2^{16} + \ldots + 1) \times 2^4 \times 3 + 2^{20} \approx 2^{25.7}$ partial decryptions. The memory requirement is small.

A 6-round attack can be made by adding one round at the beginning to construct a 4.5-round distinguisher. The plaintexts are chosen such that the inputs into the second round assume the pattern of the inputs of the 3.5-round distinguisher described above. There are four active S-boxes in the first round, and hence 16 bits of K_0 needs to be guessed. This 6-round attack would require $2^{16} \times 2^{6.4} \approx 2^{22.4}$ chosen plaintexts and $2^{16} \times 2^{25.7} \approx 2^{41.7}$ partial decryptions. The memory complexity is still small.

We can extend the attack to seven rounds by adding even another round in the end, but this attack is only better than exhaustive search for 128-bit keys. In a 7-round attack, the whole 64 bits of K_7 is needed to be guessed. After examining the key schedule for 128-bit keys, we find that 3 bits of K_6 and 58 bits of K_5 are given from guessing all of K_7. These known bits overlap in one of the bits needed from K_6 and three of the bits needed from K_5, so in total we need to guess $1 + 15 + 64 = 80$ bits of key material. The attack requires $20 \times 2^{16} \times 2^4 \approx 2^{24.3}$ chosen plaintexts and $(2^{80} + 2^{76} + \ldots + 1) \times 2^4 \times 2^{16} \approx 2^{100.1}$ partial decryptions. A total of 2^{80} bits are required to keep track of possible values for the 80 key bits, so the memory complexity is 2^{77} bytes.

3.4 Summary

The complexities of key recovery attacks on Noekeon, Serpent and PRESENT depend largely on the linear component of the round function. All 4-round attacks have been implemented on a single desktop PC. The attacks took only a few seconds to recover the last round subkey. A summary of attacks presented in this paper is shown in Table 3.

Table 3. Summary of attacks

Cipher	Rounds	Complexity		
		Data	Time	Memory
Noekeon	4	2^{17} CP	2^{26}	small
	5	$2^{20.6}$ CP	$2^{108.1}$	2^{89} bytes
Serpent	4	2^{11} CP	2^{20}	small
	5	$2^{13.6}$ CP	$2^{58.7}$	2^{44} bytes
	6	$2^{65.2}$ CP	$2^{110.7}$	2^{44} bytes
PRESENT	4	2^{5} CP	2^{13}	small
	5	$2^{6.4}$ CP	$2^{25.7}$	small
	6	$2^{22.4}$ CP	$2^{41.7}$	small
	7	$2^{24.3}$ CP	$2^{100.1}$	2^{77} bytes

4 Related Work

The applicability of the integral attack on bit-oriented ciphers was mentioned in Knudsen and Wagner's work [11]. The attack is demonstrated on the Data Encryption Standard (DES). The attack, however, works only for a very few rounds of the DES. Lucks [13] also attacked Twofish, which is not a purely byte-based cipher, with integral cryptanalysis. In Piret's thesis [14, pg 79-82], the construction of an integral distinguisher for Serpent was discussed. The distinguisher, however, does not occur with certainty and the number of rounds of the distinguisher was not explicitly mentioned.

In another work, Biryukov and Shamir [6] show how to attack a generic cipher structure which consists of non-linear and linear layers which are unknown. The technique, called the multiset attack, makes use of several multiset properties. These properties take into account whether the multiset: (1) contains arbitrary repetitions of a single value; (2) takes on all possible values; (3) contains values which occur an even number of times; (4) XOR sum equals 0; (5) has either property (2) or (3). Therefore, there is some similarities to the notations described in our work.

5 Discussion and Conclusion

In this paper, we examined the integral attack using a bit-pattern based approach. It differs from classical integral cryptanalysis in that the order of the texts in a structure becomes important, and gives the cryptanalyst a more refined notation for the texts in the structure. This information allows an attacker to gain a detailed analysis of the individual bit that propagates through the rounds. This is especially useful in analyzing the attack on ciphers that have bit-oriented round functions.

In the Noekeon document [9], it is stated that *there are no 4-round differential trails with a predicted prop ratio above* 2^{-48}. A prop ratio is the fraction of input pairs with a fixed difference that propagates into a fixed output difference. The Appendix of the Noekeon document describes that the differential trail propagates until before the non-linear component in the fourth round. In this paper, a 3.5-round distinguisher with probability 1 is discovered in which the balancedness of the structure can be retained until just before the S-box layer in the fourth round. This distinguisher is therefore comparable to the differential trail described in the document. Since our distinguisher can be constructed using just 2^{16} chosen plaintexts (as opposed to the differential attack which requires on the order of 2^{48} plaintexts), attacks using this distinguisher are much more efficient than using the best differential trail described by the authors.

The designers of Serpent have shown in their submission to the AES competition [1] that the probability of the best 5-round differential characteristic is less than 2^{-42}, so a differential attack on 5-round Serpent would require on the order of 2^{42} chosen plaintexts. The 5-round bit-pattern based integral attack described in this paper requires only $2^{13.6}$ chosen plaintexts. It should be noted that there exists better attacks on Serpent than what is reported here, the best we are aware of is a linear attack that breaks 10 rounds of Serpent with 128-bit keys [8]. The complexities of the six-round differential attack reported in [12] are comparable to the bit-pattern based integral attack described here.

In the PRESENT document [7] resistance to integral cryptanalysis is explained by noting that integral attacks are suited for ciphers with a word-wise structure, and that the design of PRESENT is bitwise. We have shown here that bit-pattern based integral cryptanalysis is indeed suited for PRESENT. Moreover, the authors show that a differential characteristic over five rounds has probability at most 2^{-20}, so a differential attack on five rounds would need on the order of 2^{20} plaintexts to succeed. In contrast, bit-pattern based integral cryptanalysis breaks five-round PRESENT with 80 chosen plaintexts.

Over a few rounds, bit-pattern based integral cryptanalysis of the three ciphers studied here is comparable to differential cryptanalysis in time complexity, but require in general much less chosen plaintext. However, differential cryptanalysis is easy to extend to more rounds whereas integral cryptanalysis can not be extended beyond a certain point. Even though the attacks do not pose a serious threat to the ciphers presented in this paper, it shows that the integral attack can still be applied to bit-oriented ciphers.

Acknowledgements

We would like to thank Adi Shamir for pointing out a related work in [6]. We also thank Leonie Simpson and anonymous referees that helped to improve this paper.

References

1. Anderson, R., Biham, E., Knudsen, L.: Serpent: A Proposal for the Advanced Encryption Standard. In: NIST AES Proposal (1998), http://www.cl.cam.ac.uk/~rja14/serpent.html
2. Biham, E., Dunkelman, O., Keller, N.: The Rectangle Attack – Rectangling the Serpent. In: Pfitzmann, B. (ed.) EUROCRYPT 2001. LNCS, vol. 2045, pp. 340–357. Springer, Heidelberg (2001)
3. Biham, E., Dunkelman, O., Keller, N.: Linear Cryptanalysis of Reduced Round Serpent. In: Matsui, M. (ed.) FSE 2001. LNCS, vol. 2355, pp. 16–27. Springer, Heidelberg (2002)
4. Biham, E., Dunkelman, O., Keller, N.: New Results on Boomerang and Rectangle Attacks. In: Daemen, J., Rijmen, V. (eds.) FSE 2002. LNCS, vol. 2365, pp. 1–16. Springer, Heidelberg (2002)
5. Biham, E., Dunkelman, O., Keller, N.: Differential-Linear Cryptanalysis of Serpent. In: Johansson, T. (ed.) FSE 2003. LNCS, vol. 2887, pp. 9–21. Springer, Heidelberg (2003)
6. Biryukov, A., Shamir, A.: Structural Cryptanalysis of SASAS. In: Pfitzmann, B. (ed.) EUROCRYPT 2001. LNCS, vol. 2045, pp. 394–405. Springer, Heidelberg (2001)
7. Bogdanov, A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J.B., Seurin, Y., Vikkelsoe, C.: PRESENT: An Ultra-Lightweight Block Cipher. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 450–466. Springer, Heidelberg (2007)
8. Collard, B., Standaert, F.-X., Quisquater, J.-J.: Improved and Multiple Linear Cryptanalysis of Reduced Round Serpent. In: Pei, D., Yung, M., Lin, D., Wu, C. (eds.) Inscrypt 2007. LNCS, vol. 4990. Springer, Heidelberg (2008)

9. Daemen, J., Peeters, M., Van Assche, G., Rijmen, V.: Nessie Proposal: NOEKEON. In: First Open NESSIE Workshop (2000), http://gro.noekeon.org/
10. Kelsey, J., Kohno, T., Schneier, B.: Amplified Boomerang Attacks Against Reduced-Round MARS and Serpent. In: Schneier, B. (ed.) FSE 2000. LNCS, vol. 1978, pp. 75–93. Springer, Heidelberg (2001)
11. Knudsen, L., Wagner, D.: Integral Cryptanalysis. In: Daemen, J., Rijmen, V. (eds.) FSE 2002. LNCS, vol. 2365, pp. 112–127. Springer, Heidelberg (2002)
12. Kohno, T., Kelsey, J., Schneier, B.: Preliminary Cryptanalysis of Reduced-Round Serpent. In: The Third Advanced Encryption Standard Candidate Conference, pp. 195–211. NIST (2000), http://csrc.nist.gov/CryptoToolkit/aes/round2/conf3/aes3conf.htm
13. Lucks, S.: The Saturation Attack – A Bait for Twofish. In: Matsui, M. (ed.) FSE 2001. LNCS, vol. 2355, pp. 1–15. Springer, Heidelberg (2002)
14. Piret, G.: Block Ciphers: Security Proofs, Cryptanalysis, Design, and Fault Attacks. PhD Thesis, Université Catholique de Louvain (2005), http://www.di.ens.fr/~piret/

Appendix

Table 4 depicts the possible values of bit patterns in a 2^4 structure. Each individual pattern has two columns to indicate that a pattern may start with bit value '0' or bit value '1'. Recall that the pattern b_0^* and b_i where $i > 0$ are not restricted to hold the values of only the XOR combinations of a_i patterns. If the patterns are composed of XOR combinations of a_i patterns, the number of occurrences of bit '0' is the same as bit '1'.

Table 4. Example of bit pattern values in a 2^4 structure

c		a_3		a_2		a_1		a_0		b_2		b_1				b_0^*					
										$\mathsf{a}_3 \oplus \mathsf{a}_2$		$\mathsf{a}_3 \oplus \mathsf{a}_2 \oplus \mathsf{a}_1$				$\mathsf{a}_2 \oplus \mathsf{a}_0$		$\mathsf{a}_3 \oplus \mathsf{a}_1 \oplus \mathsf{a}_0$			
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
0	1	0	1	0	1	0	1	1	0	0	1	0	1	0	1	1	0	1	0	0	1
0	1	0	1	0	1	1	0	0	1	0	1	1	0	1	0	0	1	1	0	1	0
0	1	0	1	0	1	1	0	1	0	0	1	1	0	1	0	1	0	0	1	0	1
0	1	0	1	1	0	0	1	0	1	1	0	1	0	1	0	1	0	0	1	0	1
0	1	0	1	1	0	0	1	1	0	1	0	1	0	1	0	0	1	1	0	0	1
0	1	0	1	1	0	1	0	0	1	1	0	0	1	1	0	1	0	1	0	0	1
0	1	0	1	1	0	1	0	1	0	1	0	0	1	1	0	0	1	0	1	1	0
0	1	1	0	0	1	0	1	0	1	1	0	1	0	1	0	0	1	1	0	0	1
0	1	1	0	0	1	0	1	1	0	1	0	1	0	1	0	1	0	0	1	0	1
0	1	1	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	0	1	0	1
0	1	1	0	0	1	1	0	1	0	1	0	0	1	1	0	1	0	1	0	1	0
0	1	1	0	1	0	0	1	0	1	0	1	0	1	0	1	1	0	1	0	0	1
0	1	1	0	1	0	0	1	1	0	0	1	0	1	0	1	0	1	0	1	1	0
0	1	1	0	1	0	1	0	0	1	0	1	1	0	1	0	1	0	0	1	0	1
0	1	1	0	1	0	1	0	1	0	0	1	1	0	1	0	0	1	1	0	0	1

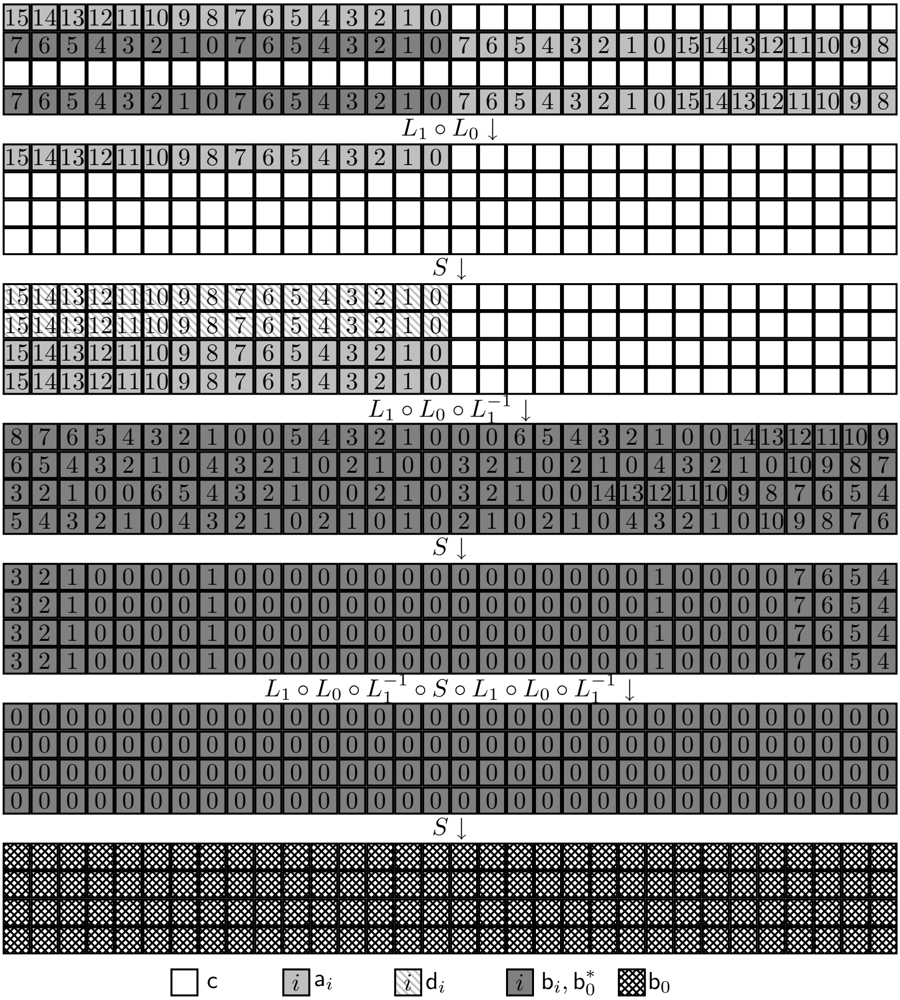


Fig. 4. The 3.5-round integral distinguisher for Noekeon. Top row of cipher block is word 0, bottom row is word 3. The rightmost column corresponds to the least significant bit (bit 0) in each word.

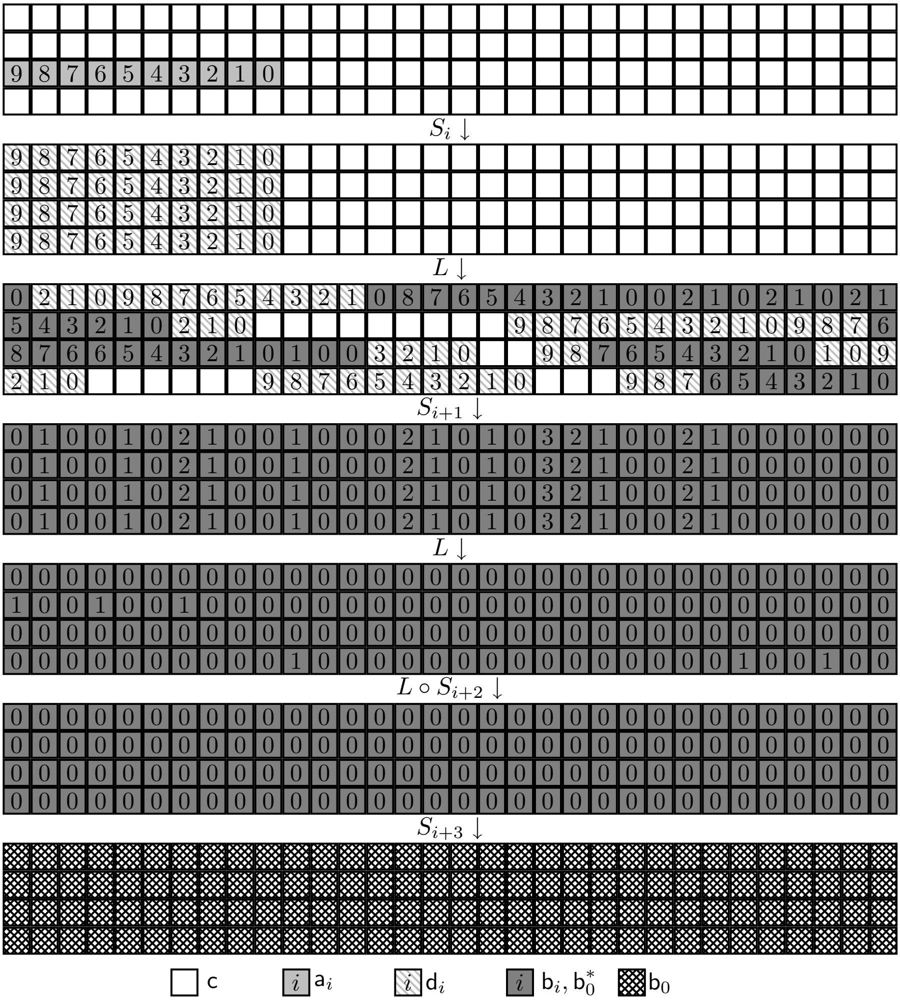


Fig. 5. The 3.5-round integral distinguisher for Serpent. Top row of cipher block is word 0, bottom row is word 3. The rightmost column corresponds to the least significant bit (bit 0) in each word.

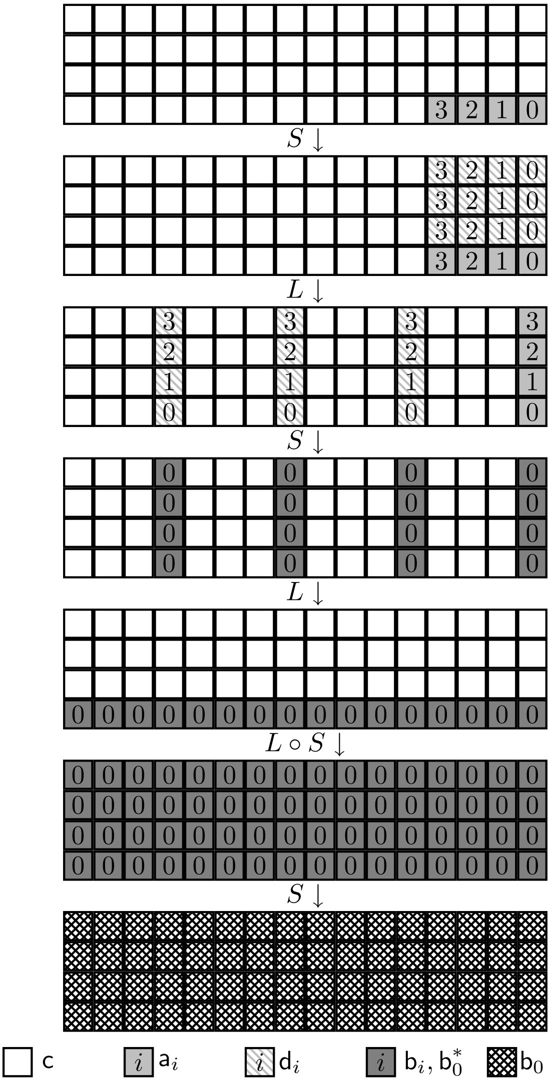


Fig. 6. The 3.5-round integral distinguisher for PRESENT. Top row of cipher block is word 0, bottom row is word 3. The rightmost column corresponds to the least significant bit (bit 0) in each word.

Experiments on the Multiple Linear Cryptanalysis of Reduced Round Serpent

B. Collard, F.-X. Standaert*, and J.-J. Quisquater

UCL Crypto Group, Microelectronics Laboratory, Louvain-la-Neuve, Belgium

Abstract. In 2004, Biryukov *et al.* presented a new theoretical framework for the linear cryptanalysis of block ciphers using multiple approximations. Although they provided first experimental results to confirm the relevance of their approach, a scope for further research was to apply this framework to other ciphers. In this paper, we present various attacks against reduced-round versions of the AES candidate Serpent. Our results illustrate that the hypotheses of Crypto 2004 hold (at least) as long as the number of approximations exploited in the linear attack are computationally tractable. But they also underline the limits and specificities of Matsui's *algorithms* 1 and 2 for the exploitation of such approximations. In particular, they show that the optimal application of *algorithm 2* requires good theoretical estimations of the approximation biases, which may be a problem when the linear hull effect is non-negligible. These results finally confirm the significant reductions of the attacks data complexity that can be obtained from multiple linear approximations.

1 Introduction

The linear cryptanalysis [10] is one of the most powerful attacks against modern block ciphers in which an adversary exploits a linear approximation of the type:

$$P[\chi_P] \oplus C[\chi_C] = K[\chi_K] \tag{1}$$

In this expression, P, C and K respectively denote the plaintext, ciphertext and the expanded key while $A[\chi]$ stands for $A_{a_1} \oplus A_{a_2} \oplus \cdots \oplus A_{a_n}$, with $A_{a_1}, \cdots, A_{a_n}$ representing particular bits of A in positions $a_1, \cdots, a_n$ (χ is usually denoted as a mask). In practice, linear approximations of block ciphers can be obtained by the concatenation of one-round approximations and such concatenations (also called characteristics) are mainly interesting if they maximize the deviation (or bias) $\epsilon = p - \frac{1}{2}$ (where p is the probability of a given linear approximation).

In its original paper, Matsui described two methods for exploiting the linear approximations of a block cipher, respectively denoted as *algorithms* 1 and 2. In the first one, given an r-round linear approximation with sufficient bias, the algorithm simply counts the number of times T the left side of Equation 1 is equal to zero for N pairs (plaintext, ciphertext). If $T > N/2$, then it assumes either $K[\chi_K] = 0$ if $\epsilon > 0$ or $K[\chi_K] = 1$ if $\epsilon < 0$ so that the experimental value

[*] Postdoctoral researcher of the Belgian Fund for Scientific Research (FNRS).

K. Nyberg (Ed.): FSE 2008, LNCS 5086, pp. 382–397, 2008.

$(T - N/2)/N$ matches the theoretical bias. If $T < N/2$, an opposite reasoning holds. For the attack to be successful, it is shown in [10] that the number of available (plaintext, ciphertext)-pairs must be proportional to $\frac{1}{\epsilon^2}$.

In the second method, a r-1-round characteristic is used and a partial decryption of the last round is performed by guessing the key bits involved in the approximation. As a consequence, all the guessed key bits can be recovered rather than the parity $K[\chi_K]$ which yields much more efficient attacks in practice. In addition, since it exploits a r-1-round approximation rather than a r-round one, it also takes advantage of a better bias which reduces the data complexity.

Among the various proposals to improve the linear cryptanalysis of block ciphers, Kaliski and Robshaw proposed in 1994 an algorithm using several linear approximations [7]. However, their method imposed a strict constraint as it requires to use only approximations implying the same bits of subkeys $K[\chi_K]$. This restricted at the same time the number and the quality of the approximations available. As a consequence, an approach removing this constraint was proposed in 2004 [2] that can be explained as follows. Let us suppose that one has access to m approximations on r block cipher rounds of the form:

$$P[\chi_P^i] \oplus C[\chi_C^i] = K[\chi_K^i] \ (1 \leq i \leq m), \tag{2}$$

and wishes to determine the value of the binary vector of parity:

$$\mathbf{Z} = (z_1, z_2, ..., z_m) = (K[\chi_K^1], K[\chi_K^2], ..., K[\chi_K^m]) \tag{3}$$

The improved algorithm associates a counter T_i with each approximation, that is incremented each time the corresponding linear approximation is verified for a particular pair (plaintext-ciphertext). As for *algorithm* 1, the values of $K[\chi_K^i]$ are determined from the experimental bias $(T_i - N/2)/N$ and the theoretical bias ϵ_i by means of a maximum likelihood rule. The extension of *algorithm* 2 to multiple approximations is similarly described in [2].

An important consequence of this work is that the theoretical data complexity of the generalized multiple linear cryptanalysis is decreased compared to the original attack. According to the authors of [2], the attack requires a number of texts inversely proportional to the capacity of the system of equations used by the adversary that is defined as: $c = 4 \cdot \sum_{i=1}^{m} \epsilon_i^2$. Therefore, by increasing this quantity (*i.e.* using more approximations), one can decrease the number of plaintext/ciphertext pairs necessary to perform a successful key recovery.

In this paper, we aim to apply the previously described cryptanalytic tools to the Advanced Encryption Standard (AES) candidate Serpent [1] in order to confirm the analysis of Crypto 2004 and put forward a number of intuitive facts about its implementation. Our results confirm the significant reductions of the attacks data complexity that can be obtained from multiple linear approximations but also their computational limitations. From a more theoretical point of view, multiple linear cryptanalysis is also the best understood technique to take advantage of the linear hull effect [11]. Therefore it allows to fill the gap between the practical [8] and provable security approaches for block ciphers.

Finally, our results underline the specificities of Matsui's *algorithms* 1 and 2 for the exploitation of multiple approximations. In particular, they show that the optimal application of *algorithm 2* requires good theoretical estimations of the approximation biases, which may be a problem when the linear hull effect is non-negligible. As a consequence, sub-optimal strategies sometimes have to be applied. By contrast, our application of *algorithm* 1 nicely follows the predictions of [2], even if the approximation biases are underestimated.

The rest of the paper is structured as follows. Section 2 refers to our linear approximation search algorithm. Section 3 describes preliminary observations on the linear cryptanalysis of Serpent and highlights the existence of a linear hull effect. Sections 4 and 5 respectively provide the experimental results of our attacks against reduced-round Serpent, using *algorithms* 1 and 2. Finally, conclusions are in Section 6 and a description of the Serpent cipher is in Appendix A.

2 Linear Approximations Search

The first step in a linear cryptanalysis attack consists in finding linear approximations of the cipher with biases as high as possible. But the problem of searching such approximations is not trivial, because of the great cardinality of the set of candidates. In 1994, Matui proposed a branch-and-bound algorithm making possible to effectively find the best approximation of the DES [12]. However, for practical reasons that are out of the scope of this paper, this method hardly applies to block ciphers with good diffusion such as the AES candidates. As a consequence, we rely on approximations found with a modified heuristic that is described in [3]. Although it does not ensure to obtain the best approximations of Serpent, it provided the best-reported ones in the open literature. Note that, given a r-round approximation found with the branch-and-bound algorithm, the first round masks and last round masks can be replaced by any other mask provided that the biases are left unchanged. Due to the properties of the Serpent S-boxes, several similar approximations can easily be generated with this technique. This allowed us to obtain large sets of approximations involving the same key bits to guess at the cost of dependencies in the linear trails[1].

3 Preliminary Observations

Prior to the investigation of multiple linear approximations, we performed experiments with single approximations. We started with *algorithm 1* of which the principle is as follows. For each plaintext-ciphertext pair, we evaluate the left part of equation 1 and increment or decrement a counter given the result. This way, the counter can be used to evaluate the experimental bias of the approximation. If the experimental and theoretical biases have the same sign, then we can presume that the parity of the subkey bits in the right part of Equation 1 is zero. Otherwise, we guess that this parity is one, so that empirical and theoretical results match. As it is suggested in [10], this heuristic relies on a maximum likelihood approach in which we choose the parity so that theoretical and practical

[1] A linear trail is a set of $r + 1$ masks describing a r-round approximation [11].

results fit well. Thus, the unknown parity can be guessed with an arbitrary high probability by computing the experimental mean of the bias and choosing the parity that minimizes the distance between theory and practice. As an illustration, we used a 4-round linear approximation of Serpent with a theoretical bias of 2^{-12} and observed its experimental value for a number of plaintext-ciphertext pairs proportional to 2^{24}. Figure 1 illustrates that the bias value becomes stable after approximately $8/\epsilon^2$ encrypted pairs. It also shows that the theoretical bias (provided by the branch-and-bound algorithm in [3]) was underestimated, which suggests that the linear hull effect is not negligible in our experiments [11]: there are several approximations with the same input/output mask that contribute to the bias in a non negligible way. This effect can cause the complexity of a linear cryptanalysis attack to be overestimated [6].

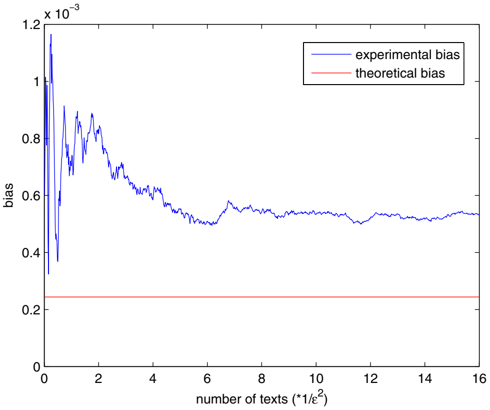


Fig. 1. Evolution of the experimental bias *w.r.t.* the number of known-plaintexts used

Next to this first experiment, we observed the behavior of 64 linear approximations with various biases, as illustrated in Figure 2. It shows that, provided a sufficient number of encrypted plaintexts, the approximations separate in two classes: the ones with positive bias and the ones with negative bias. This experiment suggests the interest of exploiting multiple linear approximations: since any of these experimental biases provide the adversary with some information on the block cipher key, it is worth trying to exploit them in an efficient way.

Following Kaliski and Robshaw [7], Biryukov *et al.* proposed a general approach to extend Matsui's linear cryptanalysis to multiple linear approximations [2]. As in the simple approximation case, an experimental bias is derived for each approximation in a distillation phase during which counters are extracted from the data. Then, in the analysis phase, an euclidian distance between the theoretical prediction and the experiment is evaluated for each possible parities of the key bits involved in the approximations. The parity minimizing this distance is guessed to be the correct one. The expectation is that the number of encrypted plaintexts required to achieve a given success rate can be reduced when the number of approximation is increased, according to the value of the capacity

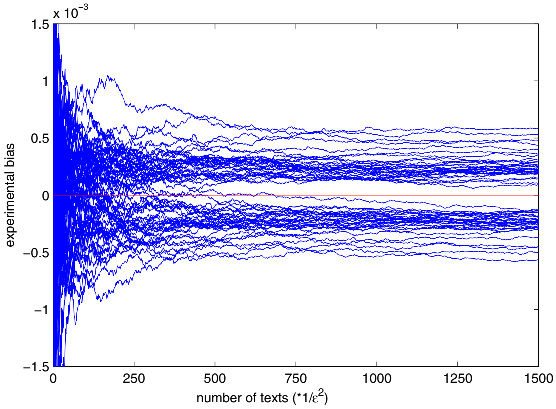


Fig. 2. Evolution of 64 experimental biases *w.r.t.* the number of known-plaintexts used

$c = 4 \cdot \sum_{i=1}^{m} \epsilon_i^2$. In the next sections, we experimentally evaluate the extent to which these expectations can be fulfilled, both for Matsui's *algorithm* 1 and 2.

4 Experimental Attacks with *Algorithm 1*

4.1 Selection of the Approximations

A significant drawback of Matsui's *algorithm* 1 compared to the second one is that the adversary does not recover master key bits, but a linear equation involving key bits in all the cipher rounds. In the context of non-linear key-scheduling algorithms, this makes the practical exploitation of the attack results difficult since it does not straightforwardly reduce the complexity of an exhaustive key search. When using multiple linear approximations, this drawback can be partially relaxed in the following way[2]. First, the best approximation provided by the branch-and-bound algorithm is selected. Then, only the input/output masks are modified in order to generate large sets of equations. Finally, the adversary progressively increases the size of its system of equations: each time he adds an equation to the system, he also checks the rank r of the corresponding matrix, indicating the number of linearly independent relations in his system. By choosing the system of equations such that the independencies between the equations only relate to meaningful key bits, the adversary ends up with an exploitable information on the cipher key. For example, if the adversary only modifies the input masks to generate a system of the form:

$$P[\chi_P^i] \oplus C[\chi_C^i] = K[\chi_K^i] \ (1 \leq i \leq m), \tag{4}$$

he can recover first round key bits. Only one bit (corresponding to the non-variable part of the trail in the system) has to be guessed additionally. As an

[2] Of course, it remains that *algorithm* 1 uses a r-round approximation compared to a $r-1$-round approximation in *algorithm* 2 which increases its data complexity.

illustration, we performed attacks against 4-rounds of Serpent, using 64 approximations such that the resulting system of equations has rank $r = 10$.

4.2 Attacks Results

Figure 3 depicts the evolution of the distance between the theoretical and experimental biases, for various values of the parity guess and number of encrypted plaintexts. The correct key candidate is expected to minimize this distance which is verified in practice: $32/c$ encrypted plaintexts are sufficient to uniquely determine the correct parity guess. As expected, increasing the number of encrypted plaintexts improves the confidence (or reduces the noise) in the attack result. For example, Figure 4 illustrates the result of a similar attack when $4096/c$ encrypted plaintexts are provided to the adversary. Interestingly, the attack result has a very regular structure underlining the impact of the Hamming distance between different key candidates: close keys have close biases. However, such figure does not tell us (or quantify) how much the exploitation of multiple approximations allowed reducing the attack data complexity.

For this purposes, we ran another set of experiments in which we computed the gain of the attack. As defined in [2], *if an attack is used to recover an n-bit*

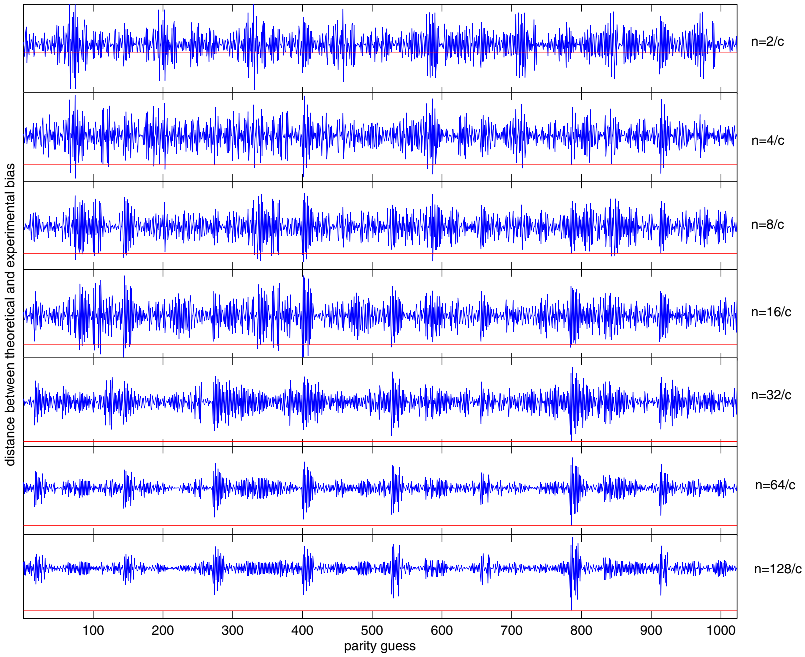


Fig. 3. Evolution of the distance between the theoretical and experimental biases *w.r.t.* the parity guess when the number of encrypted plaintexts increases (64 4-rounds approximations with a capacity of $5.25 \cdot 10^{-6}$). The horizontal line in each graph indicates this distance for the correct parity guess. The scale is the same in each figure.

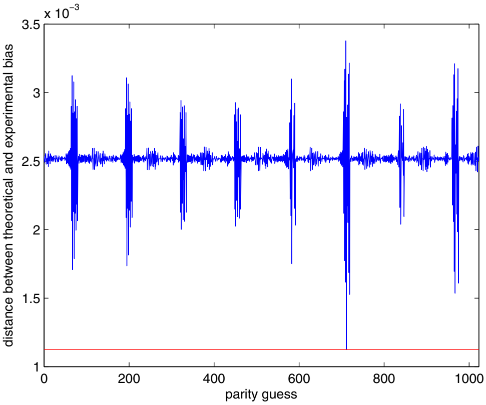


Fig. 4. Evolution of the distance between the theoretical and experimental biases (same as Figure 3) *w.r.t.* the parity guess when using $4096/c$ encrypted plaintexts

key and is expected to return the correct key after having checked on the average M candidates, then the gain of the attack, expressed in bits, is defined as:

$$\gamma = -log_2 \frac{2 \cdot M - 1}{2^n} \tag{5}$$

Intuitively, it is a measure of the remaining workload (or number of key candidates to test) after a cryptanalysis has been performed. In the context of multiple linear cryptanalysis attacks, the gain is simply determined by the position of the correct key (or parity) candidate in the weighted list of candidates obtained from the analysis phase. For example, if an attack is used to recover 8 key bits and the correct key candidate is the most likely (*resp.* second most likely), it has a gain of 8 bits (*resp.* 6.42 bits). Importantly, when *algorithm* 1 is used, the maximum gain of an attack depends on the rank of its systems of equations.

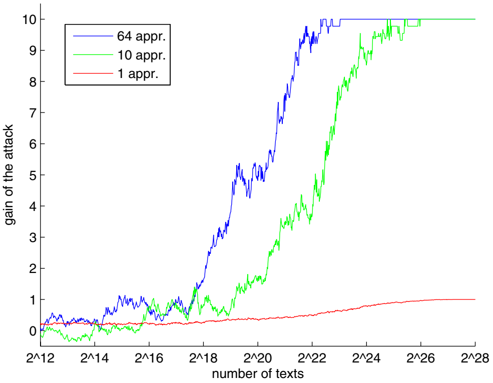


Fig. 5. Evolution of the gain with respect to the number of encrypted plaintexts

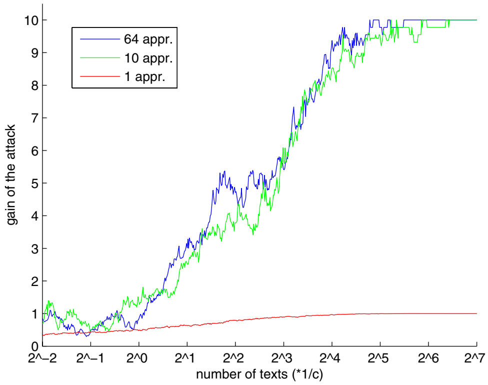


Fig. 6. Evolution of the gain *w.r.t.* the number of plaintexts normalized by the capacity

In Figure 5, the gain of three attacks are given with respect to the data complexity. The first attack (in red) recovers one bit of parity using only one approximation (*i.e.* it is a simple linear cryptanalysis). The second attack (in green) uses 10 approximations and recovers up to 10 parity bits, while the third attack (in blue) recovers 10 parity bits using 64 linearly dependent approximations. As expected, the gain obtained using 64 approximations increases about 8 times faster than with 10 approximations. This example shows the flexibility offered by multiple approximations when *algorithm* 1 is applied: they can be used both to get a better gain for a fixed number of plaintext or to get a lower data complexity for a fixed gain. This observation is even better quantified in Figure 6 where the evolution of the gain is given according to the number of encrypted plaintexts normalized by the capacity (*i.e.* the number of plaintexs divided by the joint capacity of the approximations used in the attack). It clearly illustrates the tradeoff between attack complexity and gain. It also confirms that $N \propto 1/c$ is the number of plaintexts required for the attack to reach its maximum gain.

4.3 Gain Versus Success Rate and Further Insights

Let us introduce the following definition:

Definition 1 (success rate). *The success rate of an attack using n approximations (for a given number of plaintexts/ciphertexts) is the number of parity bits guessed correctly among the n parities derived from the distance between the experimental and theoretical bias values.*

Figure 7 represents the evolution of the gain and of the success rate when the number of encrypted plaintexts increases. Interestingly, we can see that the gain increases much faster than the success rate. For example, after about 2^{23} encrypted plaintexts, the gain of the attack reaches its maximum, while the success rate only equals 0.8 at this point. This is a consequence of the linear dependancies between the approximations. Suppose we are given m linear approximations:

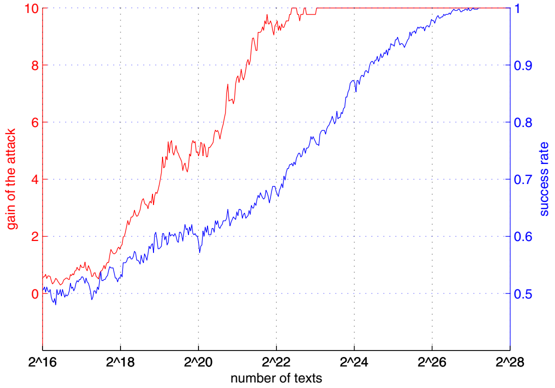


Fig. 7. Evolution of the gain and success rate *w.r.t.* the number of encrypted plaintexts

$$P[\chi_P^i] \oplus C[\chi_C^i] = K[\chi_K^i] \ (1 \leq i \leq m), \tag{6}$$

Such that the following relation holds (case of dependant text masks):

$$\chi_P^1 \oplus \chi_P^2 \oplus ... \oplus \chi_P^m = \chi_C^1 \oplus \chi_C^2 \oplus ... \oplus \chi_C^m \tag{7}$$

Approximation m is linearly dependant in the sense that no additionnal information is given in the deterministic case (i.e. if the approximations hold with probability 1). However, in the probabilistic case, some information can still be extracted (as shown in [13]), as the bias of the $m-th$ approximation is not necessarily related to the bias of the $m-1$ first. When performing multiple linear approximations cryptanalysis, the left part of each approximation is evaluated for a large number of plaintext-ciphertexts pairs and then the parity of the right part (involving subkey bits) is choosen so as to minimize the distance between experimental and theoretical bias. However, some of the parity guess might be wrong in which case the system of equations (where p^i is the parity guess for approximation i):

$$\begin{cases} K[\chi_K^1] = p^1 \\ K[\chi_K^2] = p^2 \\ \quad ... \\ K[\chi_K^m] = p^m \end{cases}$$

can be inconsistent given the linear dependancies, and one or more parity guess must be changed. This can be verified only if there are linear dependancies between the approximations. As the consistency check can be performed before the exhaustive search for the remaining unknown bits, this increases the gain of the attack. Intuitively, this shows that using more approximations than the rank of the system in an attack provides an effect similar to an error correcting code: some parity candidates can be rejected a-priori. By contrast, the success rate of an attack is expected to remain the same when the number of approximations increases (provided that the theoretical biases of each approximation are equal).

As an illustration, we can study the relation between the success rate and the gain of an attack. Suppose that at least n' out of the n approximation parity are guessed correctly. Obviously, the succes rate is higher than n'/n. In order to recover the key (and evaluate the gain), we must generate a list of candidates from the value of the parity bits and then try each candidate until the correct one is found. This can be done using the following strategy:

- Choose the first candidate so as to minimize the euclidian distance between theoretical and experimental bias.
- Assume one guess is incorrect; choose one parity bit and take its complement; try the $\binom{n}{1}$ possible candidates;
- Assume two guesses are incorrect; choose two parity bits and take their complements; try the $\binom{n}{2}$ possible candidates;
- ...
- Assume $n-n'$ guesses are incorrect; choose $n-n'$ parity bits and take their complements; try the $\binom{n}{n-n'}$ possible candidates;

After $n - n'$ steps, we have necessarily found the correct candidate as there is maximum $n - n'$ wrong guesses, thus the gain of the attack equals:

$$\gamma = -log_2\left(\frac{\sum_{i=0}^{n-n'} \binom{n}{i}}{2^n}\right) \tag{8}$$

In this equation, we implicitly assume the independence between the approximations (*i.e.* we assume the maximum gain can be n). However, experiments using up to 416 approximations (including 15 linearly independent ones) show that this prediction fits reasonably well even when the approximation are not independent, as long as the gain does not saturate to its maximum value (see Figure 8). We observe that for a given success rate, the gain of the attack increases with

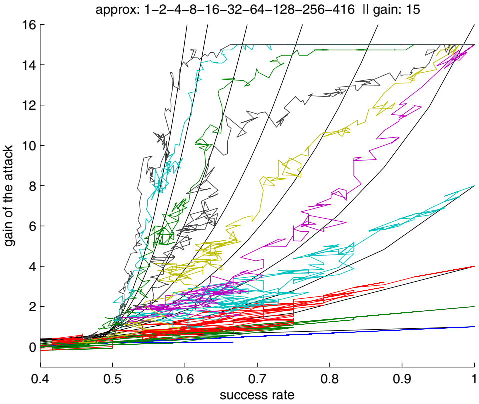


Fig. 8. Evolution of the gain *w.r.t.* the success rate for various number of approximations. This number ranges from 1 to 416 when moving from the bottom curves to the top curves. The black smooth curves are the theoretical predictions.

the number of approximations. This example highlights (from a different point of view) the advantage of multiple linear approximations compared to single linear cryptanalysis in which the success rate is equivalent to the gain.

5 Experimental Attacks with *Algorithm 2*

In this section, we perform experimental attacks against 5-round Serpent using multiple linear approximations with *Algorithm 2*. It allows the adversary to recover subkey bits in the first/last round of the cipher as follows. An attack against r cipher rounds deals with a linear approximation of the $r-1$ last/first rounds. For each plaintext-ciphertext pair and subkey candidate, the ciphertext is then partially en/decrypted with the subkey candidate, and the approximation is evaluated for the partial en/decryption. A counter indexed by the keyguess value is incremented/decremented according to the parity of the evaluation. For a wrong candidate, the partial en/decryption is expected to produce a random output, thus leading to an null experimental bias (meaning that its statistical evaluation is sufficiently close to zero). For the correct key guess, the experimental bias is expected to converge toward the theoretical bias. In order to speed-up the computations, we used the FFT trick proposed in [4].

5.1 Differences between *Algorithms* 1 and 2

Compared with *algorithm 1*, the exploitation of multiple linear approximations with *algorithm* 2 faces an additional problem that we detail in this section. In multiple linear cryptanalysis, an adversary has a system of m linear approximations. Each approximation has a theoretical value for its bias ϵ_i and the adversary additionally obtains an experimental value for this bias ϵ_i^*. When *algorithm* 1 is used, this experimental value is used to minimize the Euclidean distance:

$$\min_g \sum_{i=1}^{m} (\epsilon_i - (-1)^{g(i)} \cdot \epsilon_i^*)^2, \tag{9}$$

where g is the parity guess of the linear approximations. But when *algorithm* 2 is used, this experimental bias also depends on the round subkey that is used to perform the first (or last) round partial decryption: $\epsilon_{i,k}^*$. Therefore, the following Euclidean distance has to be computed:

$$\min_k \Big(\min_g \sum_{i=1}^{m} (\epsilon_i - (-1)^{g(i)} \cdot \epsilon_{i,k}^*)^2 \Big) \tag{10}$$

The practical consequence of these different conditions can be explained as follows. While the condition in Equation 9 leads to a correct value for the guess, even if the theoretical bias values are underestimated, the condition in Equation 10 only leads to a correct key candidate if a good theoretical estimation of these biases is available. This is due to the key dependencies of the experimental biases in *algorithm* 2. Unfortunately, in the context of the Serpent algorithm investigated in this paper, it has been shown in Section 3 that these theoretical values are not accurate, due to the linear hull effect. As a consequence, the framework of [2] cannot be straightforwardly applied in our context. This is in contrast, *e.g.* with the DES, where such a linear hull effect is negligible [6].

5.2 Attack Results

The usual solution to overcome this problem is to look at the maximum experimental bias values. For example, when multiple approximations are considered, a vector of experimental biases can be derived for each approximation. Then the average is taken over all the approximations and its maximum value is expected to indicate the correct key candidate. But the theoretical framework of Crypto 2004 is not applied anymore and such an approach is more closely related to the experiments of Kaliski and Robshaw [7]. As an illustration, Figure 9 illustrates the evolution of the experimental biases averaged over 32 4-rounds approximations, for different numbers of plaintext/ciphertext pairs and 12 bits of keyguess. Figure 10 shows a comparison between simple and multiple linear cryptanalysis.

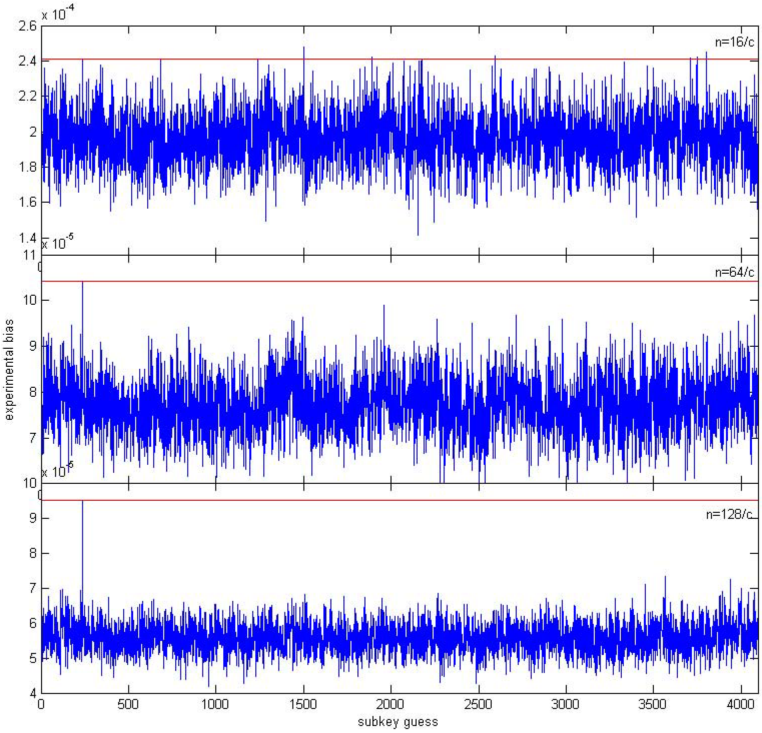


Fig. 9. Evolution of the experimental bias averaged over 32 4-rounds approximations for each guess of the 2^{12} key guesses, when the number of pairs increases ($c = 2.08 \cdot 10^{-7}$).

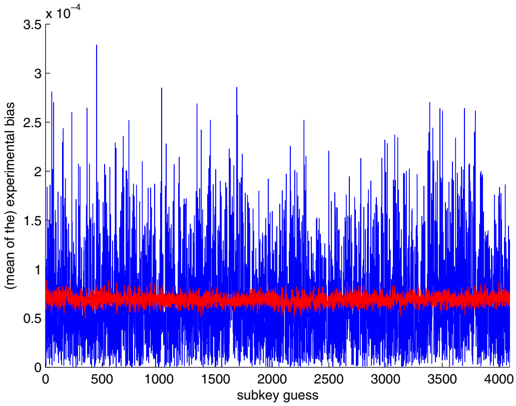


Fig. 10. Comparison of the noise levels between simple and multiple linear cryptanalysis (104 approximations vs. 1 approximation, n=64/c)

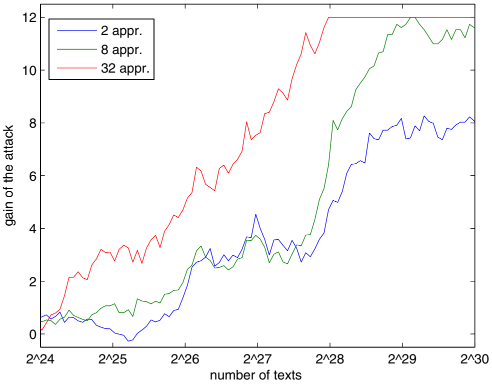


Fig. 11. Evolution of the gain of *algorithm 2* w.r.t. the number of encrypted plaintexts

This latter picture illustrates that multiple linear approximations still allow reducing the noise level in the modified attacks which improves their efficiency.

On the other hand, increasing the number of linear approximations does not involve reductions of the data complexity according to the capacity values as in the previous section. This is caused by the modified analysis phase, in which the exploitation of the information provided by the different approximations is not optimal anymore. This fact can be emphasized by investigating the gain of the attacks, for different number of approximations. For example, Figure 11 illustrates how the gain of three attacks with respectively 2, 8 and 32 approximations (having the same individual biases) increases. But the benefits are not as spectacular is in the context of *algorithm 1*. Namely, the 32-approximations gain is not increasing 16 times faster than when 2 approximations are used. Compared to the attack results with *algorithm* 1 (*e.g.* in Figure 4), it is finally interesting to notice that Figures 9 and 10 show no particular structure in the biases distribution. This is due to the partial en/decryption of one round that cancels the effects caused by close keys in the Hamming distance sense.

6 Conclusion and Further Works

This paper presented experimental results of multiple linear cryptanalysis attacks against reduced-round versions of the block cipher Serpent. It allowed us to highlight the following observations:

1. The hypotheses stated in [2] about the possible influence of dependencies (between the masks or linear trails) generally appear to be reasonably fulfilled, even for approximations of which a large part of the trail is identical.
2. Our experiments only considered a limited number of approximations. Dependencies effects could appear with more approximations. Note that the number of exploitable approximations is limited anyway, for computational reasons: because of the approximation matrix rank with *algorithm* 1 and because of the amount of partial en/decryptions to perform in *algorithm* 2.
3. By contrast with previous experiments against the DES, we observed a significant linear hull effect, with the following consequences:
   (a) Optimal attacks using Matsui's *algorithm* 1 closely followed the data complexities predicted with the capacity value (defined in [2]), even if the theoretical values of the approximation biases were underestimated.
   (b) Optimal attacks using Matsui's *algorithm* 2 did not lead to successful key recoveries because of the lack of good theoretical estimations of the bias values. Modified heuristics allowed us to take advantage of multiple approximations. But the improvement of the modified attack complexity is not following the predictions of the capacity values.
4. More generally, the analysis of Crypto 2004 leads to meaningful results as long as the branch-and-bound algorithm used to derive the linear approximations provides the adversary with the best possible biases.

In practice, our experiments finally confirmed the significant improvement of multiple linear cryptanalysis attacks compared to Matsui's original attack. Open questions include the optimal exploitation of multiple approximations using *algorithm* 2 when good estimations of the bias values are not available or the extension of these experiments towards more cipher rounds, *e.g.* using [9].

Description of the Approximations

A detailed description of the linear approximations used in our experiments is available at the following address:
http://www.dice.ucl.ac.be/∼fstandae/PUBLIS/50b.zip

Acknowledgements

This work is partially supported by the Walloon Region under the project Nanotic-Cosmos and by the Belgian Interuniversity Attraction Pole P6/26 BCRYPT.

References

1. Anderson, R., Biham, E., Knudsen, L.: Serpent: A Proposal for the Advanced Encryption Standard. In: The Proceedings of the First Advanced Encryption Standard (AES) Conference, Ventura, CA (1998)
2. Biryukov, A., De Cannière, C., Quisquater, M.: On Multiple Linear Approximations. In: Franklin, M. (ed.) CRYPTO 2004. LNCS, vol. 3152, pp. 1–22. Springer, Heidelberg (2004)
3. Collard, B., Standaert, F.-X., Quisquater, J.-J.: Improved and Multiple Linear Cryptanalysis of Reduced Round Serpent. In: Pei, D., Yung, M., Lin, D., Wu, C. (eds.) InsCrypt 2007. LNCS, vol. 4990, pp. 47–61. Springer, Heidelberg (2008)
4. Collard, B., Standaert, F.-X., Quisquater, J.-J.: Improving the Time Complexity of Matsui's Linear Cryptanalysis. In: Nam, K.-H., Rhee, G. (eds.) ICISC 2007. LNCS, vol. 4817, pp. 77–88. Springer, Heidelberg (2007)
5. Daemen, J., Rijmen, V.: The Wide-Trail Strategy. In: Honary, B. (ed.) Cryptography and Coding 2001. LNCS, vol. 2260, pp. 222–238. Springer, Heidelberg (2001)
6. Junod, P.: On the Complexity of Matsui's Attack. In: Vaudenay, S., Youssef, A.M. (eds.) SAC 2001. LNCS, vol. 2259, pp. 199–211. Springer, Heidelberg (2001)
7. Kaliski, B.S., Robshaw, M.J.B.: Linear Cryptanalysis using Multiple Approximations. In: Desmedt, Y.G. (ed.) CRYPTO 1994. LNCS, vol. 839, pp. 26–39. Springer, Heidelberg (1994)
8. Knudsen, L.R.: Practically Secure Feistel Ciphers. In: Anderson, R. (ed.) FSE 1993. LNCS, vol. 809, pp. 211–221. Springer, Heidelberg (1994)
9. Kumar, S., Paar, C., Pelzl, J., Pfeiffer, G., Schimmler, M.: Breaking Ciphers with COPACOBANA - A Cost-Optimized Parallel Code Breaker. In: Goubin, L., Matsui, M. (eds.) CHES 2006. LNCS, vol. 4249, Springer, Heidelberg (2006)
10. Matsui, M.: Linear cryptanalysis method for DES cipher. In: Helleseth, T. (ed.) EUROCRYPT 1993. LNCS, vol. 765, pp. 386–397. Springer, Heidelberg (1994)
11. Nyberg, K.: Linear Approximations of Block Ciphers. In: De Santis, A. (ed.) EUROCRYPT 1994. LNCS, vol. 950, pp. 439–444. Springer, Heidelberg (1995)
12. Matsui, M.: On Correlation Between the Order of S-boxes and the Strength of DES. In: De Santis, A. (ed.) EUROCRYPT 1994. LNCS, vol. 950, pp. 366–375. Springer, Heidelberg (1995)
13. Murphy, S.: The Independence of Linear Approximations in Symmetric Cryptology. IEEE Transactions on Information Theory 52, 5510–5518 (2006)

A The Serpent Algorithm

The Serpent block cipher was designed by Ross Anderson, Eli Biham and Lars Knudsen [1]. It was an Advanced Encryption Standard candidate, finally rated just behind the AES Rijndael. Serpent has a classical SPN structure with 32 rounds and a block width of 128 bits. It accepts keys of 128, 192 or 256 bits and is composed of the following operations:

- an initial permutation IP,
- 32 rounds, each of them built upon a subkey addition, a passage through 32 S-boxes and a linear transformation L (excepted the last round, where the linear transformation is not applied),
- a final permutation FP.

In each round R_i, only one S-box is used 32 times in parallel. The cipher uses 8 distinct S-boxes S_i $(0 \leq i \leq 7)$ successively along the rounds and consequently, each S-box is used in exactly four different rounds. Finally, the linear diffusion transform is entirely defined by *XOR*s ($\oplus$), *rotations* ($\lll$) and left *shifts* ($\ll$). Its main purpose is to maximize the avalanche effect within the cipher. If one indicates by X_0, X_1, X_2, X_3 the $4 \cdot 32$ bits at the input of the linear transformation, it can be defined by the following operations:

$$\begin{array}{rl}
\text{input} = & X_0, X_1, X_2, X_3 \\
\hline
X_0 = & X_0 \lll 13 \\
X_2 = & X_2 \lll 3 \\
X_1 = & X_1 \oplus X_0 \oplus X_2 \\
X_3 = & X_3 \oplus X_2 \oplus (X_0 \ll 3) \\
X_1 = & X_1 \lll 1 \\
X_3 = & X_3 \lll 7 \\
X_0 = & X_0 \oplus X_1 \oplus X_3 \\
X_2 = & X_2 \oplus X_3 \oplus (X_1 \ll 7) \\
X_0 = & X_0 \lll 5 \\
X_2 = & X_2 \lll 22 \\
\hline
\text{output} = & X_0, X_1, X_2, X_3
\end{array}$$

Impossible Differential Cryptanalysis of CLEFIA

Yukiyasu Tsunoo[1], Etsuko Tsujihara[2], Maki Shigeri[3], Teruo Saito[3],
Tomoyasu Suzaki[1], and Hiroyasu Kubo[3]

[1] NEC Corporation, 1753, Shimonumabe, Nakahara, Kawasaki 211-8666, Japan
{tsunoo@BL,t-suzaki@cb}.jp.nec.com

[2] Y.D.K.Co., Ltd., 1288, Oshitate, Inagi-Shi, Tokyo 206-0811, Japan
etsuko-t@ghn.ydkinc.co.jp

[3] NEC Software Hokuriku, Ltd., 1,Anyoji, Hakusan, Ishikawa 920-2141, Japan
{m-shigeri@pb,t-saito@qh,h-kubo@ps}.jp.nec.com

Abstract. This paper reports impossible differential cryptanalysis on the 128-bit block cipher CLEFIA that was proposed in 2007, including new 9-round impossible differentials for CLEFIA, and the result of an impossible differential attack using them. For the case of a 128-bit key, it is possible to apply the impossible differential attack to CLEFIA reduced to 12 rounds. The number of chosen plaintexts required is $2^{118.9}$ and the time complexity is 2^{119}. For key lengths of 192 bits and 256 bits, it is possible to apply impossible differential attacks to 13-round and 14-round CLEFIA. The respective numbers of chosen plaintexts required are $2^{119.8}$ and $2^{120.3}$ and the respective time complexities are 2^{146} and 2^{212}. These impossible differential attacks are the strongest method for attacking reduced-round CLEFIA.

Keywords: block cipher, CLEFIA, diffusion switching mechanism, generalized Feistel structure, impossible differential cryptanalysis.

1 Introduction

Differential attacks [2] and linear attacks [3] are the most common methods of attack applied to block ciphers. Guaranteeing security against differential attacks and linear attacks is an important problem in the design of block ciphers. One known method of evaluating security against such attacks uses the minimum number of active S-boxes. Shirai et al. proposed in 2004 the diffusion switching mechanism (DSM), a method of designing a Feistel structure block cipher that can guarantee a large minimum number of active S-boxes [4,5]. In 2007, CLEFIA, a 128-bit block cipher designed using DSM, was proposed [6]. The designers of CLEFIA adopted a four-branch generalized Feistel structure to achieve both a small implementation size and high speed. The generalized Feistel structure tends to require more rounds to guarantee security than does an ordinary Feistel structure, but CLEFIA can guarantee resistance to differential attacks and linear attacks with a small number of rounds because of the use of DSM.

The impossible differential attack [1] is a method that was first applied against Skipjack to reject wrong key candidates by using input difference and output

K. Nyberg (Ed.): FSE 2008, LNCS 5086, pp. 398–411, 2008.

difference pairs whose probabilities are zero (impossible differentials). Impossible differentials that are dependent on the basic structure of the data processing part are often used, and this method is a particular threat to the generalized Feistel structure. Since CLEFIA is a generalized Feistel structure, the impossible differential attack is an effective attack against CLEFIA. According to the designers, an evaluation of CLEFIA with respect to an impossible differential attack [6,7] shows that there are 9-round impossible differentials in CLEFIA, and for a 128-bit key, a 10-round impossible differential attack is possible. For key lengths of 192 bits and 256 bits, 11-round and 12-round impossible differential attacks are possible.

In this paper, we show that there are previously unknown 9-round impossible differentials in CLEFIA and report the result of impossible differential attacks using those impossible differentials. These impossible differentials exist in structures that are designed using DSM. In the impossible differential attacks on CLEFIA described in this paper, 12-round CLEFIA can be broken for a 128-bit key. For key lengths of 192 bits and 256 bits, impossible differential attacks are respectively possible for 13-round and 14-round CLEFIA.

There have been no reports on the cryptanalysis of CLEFIA other than the evaluation by the designers. Accordingly, the strong attack method for CLEFIA up to now is the differential attack and linear attack described in the designers' evaluation, which shows the possibility of 12-round, 13-round, and 14-round attack for the respective key lengths of 128 bits, 192 bits, and 256 bits. Nevertheless, these results are values for guaranteeing security with respect to differential attacks or linear attacks; the numbers of rounds for establishing actual differential attacks or linear attacks are probably smaller. Accordingly, the impossible differential attacks described in this paper are the result for the most number of rounds as an actual attack method on CLEFIA.

In this paper, we describe the CLEFIA structure in Sect. 2, explain the newly discovered impossible differentials and present attack procedures against CLEFIA using those differentials in Sect. 3. Section 4 concludes this paper.

2 Description of CLEFIA

2.1 Notation

We use the following notation in this paper.

$a_{(b)}$	b is the bit length of a If the bit length of a is known, (b) is omitted.
$a \mid b$	The concatenation of a and b
$[a, b]$	The vector representation of $a \mid b$
${}^t a$	Transposition of vector a or matrix a
$[x^{\{i,0\}}, x^{\{i,1\}}, x^{\{i,2\}}, x^{\{i,3\}}]$	i-round output data, $x^{\{i,j\}} \in \{0,1\}^{32}$ The plaintext is $[x^{\{0,0\}}, x^{\{0,1\}}, x^{\{0,2\}}, x^{\{0,3\}}]$.
$[C^{\{i,0\}}, C^{\{i,1\}}, C^{\{i,2\}}, C^{\{i,3\}}]$	The i-round CLEFIA ciphertext

$a \oplus b$	Bit-wise exclusive OR of a and b (addition over $\mathrm{GF}(2^n)$)
Δa	Difference for a (difference over $\mathrm{GF}(2^n)$)
$w_b(a)$	For an $8n$-bit string $a = a_{0(8)} \mid a_{1(8)} \mid \ldots \mid a_{n-1(8)}$, $w_b(a)$ denotes the number of non-zero a_is.
$B(P)$	Branch number for function P $B(P) = min_{a \neq 0}\{w_b(a) + w_b(P(a))\}$

2.2 Structure

In this section, we explain only the data processing part of CLEFIA.

CLEFIA is a block cipher that has a block length of 128 bits and key lengths of 128, 192, and 256 bits.

The data processing part is a four-branch generalized Feistel structure with two parallel F functions (F_0, F_1) per round. The number of respective rounds r for 128-bit, 192-bit and 256-bit keys are 18, 22 and 26. The encryption function ENC_r generates 128-bit ciphertext from 128-bit plaintext, $2r$ 32-bit round keys $(RK_{0(32)}, \ldots, RK_{2r-1(32)})$, and four 32-bit whitening keys $(WK_0, \ldots, WK_3)$. The structure of the encryption function ENC_r is shown in Fig. 1. ENC_r is defined as follows.

```
$ENC_r$:
Step 1.   $T_0 \mid T_1 \mid T_2 \mid T_3 \leftarrow x^{\{0,0\}} \mid (x^{\{0,1\}} \oplus WK_0) \mid x^{\{0,2\}} \mid (x^{\{0,3\}} \oplus WK_1)$
Step 2.   For $i$=0 to $r-1$ do the following:
Step 2.1. $T_1 \leftarrow T_1 \oplus F_0(RK_{2i}, T_0)$, $T_3 \leftarrow T_3 \oplus F_1(RK_{2i+1}, T_2)$
Step 2.2. $T_0 \mid T_1 \mid T_2 \mid T_3 \leftarrow T_1 \mid T_2 \mid T_3 \mid T_0$
Step 3.   $C^{\{r,0\}} \mid C^{\{r,1\}} \mid C^{\{r,2\}} \mid C^{\{r,3\}} \leftarrow T_3 \mid (T_0 \oplus WK_2) \mid T_1 \mid (T_2 \oplus WK_3)$
```

The two F functions, F_0 and F_1, have 32-bit data x and 32-bit key RK as input; they output the 32-bit data y. F_0 is defined as follows.

```
$F_0$:
Step 1. $T \leftarrow RK \oplus x$
Step 2. Let $T = T_{0(8)} \mid T_{1(8)} \mid T_{2(8)} \mid T_{3(8)}$
        $T_0 \leftarrow S_0(T_0)$, $T_1 \leftarrow S_1(T_1)$, $T_2 \leftarrow S_0(T_2)$, $T_3 \leftarrow S_1(T_3)$
Step 3. Let $y = y_{0(8)} \mid y_{1(8)} \mid y_{2(8)} \mid y_{3(8)}$
        ${}^t[y_0, y_1, y_2, y_3] = M_0 \, {}^t[T_0, T_1, T_2, T_3]$
```

F_1 is defined by replacing the terms in F_0 as follows: S_0 is replaced with S_1, S_1 with S_0, and M_0 with M_1. The structures of F_0 and F_1 are shown in Fig. 2.

S_0 and S_1 are non-linear 8-bit S-boxes.

The two matrices M_0 and M_1 are defined as

$$M_0 = \begin{pmatrix} \mathrm{0x01} & \mathrm{0x02} & \mathrm{0x04} & \mathrm{0x06} \\ \mathrm{0x02} & \mathrm{0x01} & \mathrm{0x06} & \mathrm{0x04} \\ \mathrm{0x04} & \mathrm{0x06} & \mathrm{0x01} & \mathrm{0x02} \\ \mathrm{0x06} & \mathrm{0x04} & \mathrm{0x02} & \mathrm{0x01} \end{pmatrix}, \quad M_1 = \begin{pmatrix} \mathrm{0x01} & \mathrm{0x08} & \mathrm{0x02} & \mathrm{0x0}a \\ \mathrm{0x08} & \mathrm{0x01} & \mathrm{0x0}a & \mathrm{0x02} \\ \mathrm{0x02} & \mathrm{0x0}a & \mathrm{0x01} & \mathrm{0x08} \\ \mathrm{0x0}a & \mathrm{0x02} & \mathrm{0x08} & \mathrm{0x01} \end{pmatrix}.$$

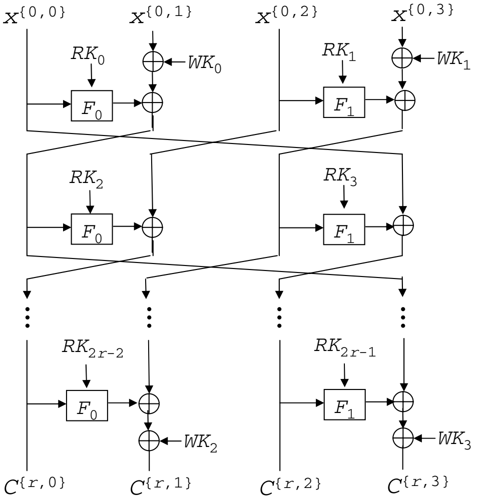


Fig. 1. Encryption function ENC_r

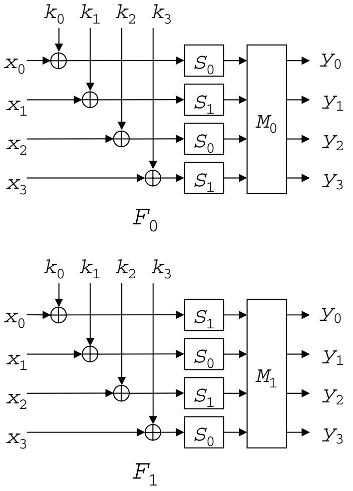


Fig. 2. Functions F_0 and F_1

The multiplications between these matrices and vectors are performed in $GF(2^8)$ defined by the primitive polynomial $z^8 + z^4 + z^3 + z^2 + 1$.

M_0 and M_1 satisfy

$$B(M_0) = B(M_1) = 5,\ B(M_0 \,|\, M_1) = B({}^tM_0^{-1} \,|\, {}^tM_1^{-1}) = 5.$$

3 Impossible Differential Attacks on CLEFIA

In this section, we present the new 9-round impossible differentials in Sect. 3.1, and explain the procedure for using those impossible differentials to attack CLEFIA in Sect. 3.2 and subsequent sections.

3.1 Nine-Round Impossible Differentials of CLEFIA

The following two new 9-round impossible differentials are found in CLEFIA,

$$[0_{(32)}, 0_{(32)}, 0_{(32)}, \alpha_{in(32)}] \not\rightarrow_{9r} [0_{(32)}, 0_{(32)}, 0_{(32)}, \alpha_{out(32)}]$$
$$[0_{(32)}, \alpha_{in(32)}, 0_{(32)}, 0_{(32)}] \not\rightarrow_{9r} [0_{(32)}, \alpha_{out(32)}, 0_{(32)}, 0_{(32)}]$$

where α_{in} and α_{out} are the differences shown in Table 1. The $X_{(8)}$ and $Y_{(8)}$ in α_{in} and α_{out} are arbitrary non-zero values. These impossible differentials are entirely different from the impossible differentials found by the designers. The first impossible differential is represented in Fig. 3.

Here, we prove that where $\alpha_{in} = [0_{(8)}, 0_{(8)}, 0_{(8)}, X_{(8)}]$, and $\alpha_{out} = [Y_{(8)}, 0_{(8)}, 0_{(8)}, 0_{(8)}]$, the probability of $[0_{(32)}, 0_{(32)}, 0_{(32)}, \alpha_{in}]$ occurring nine rounds after $[0_{(32)}, 0_{(32)}, 0_{(32)}, \alpha_{out}]$ is zero, which is to say that $[0, 0, 0, \alpha_{in}] \not\rightarrow_{9r} [0, 0, 0, \alpha_{out}]$ is an impossible differential. Other impossible differentials can be proven in the same way.

Table 1. Differential values for α_{in} and α_{out}

α_{in}	α_{out}
$[0_{(8)}, 0_{(8)}, 0_{(8)}, X_{(8)}]$	$[0_{(8)}, 0_{(8)}, Y_{(8)}, 0_{(8)}], [0_{(8)}, Y_{(8)}, 0_{(8)}, 0_{(8)}], [Y_{(8)}, 0_{(8)}, 0_{(8)}, 0_{(8)}]$
$[0_{(8)}, 0_{(8)}, X_{(8)}, 0_{(8)}]$	$[0_{(8)}, 0_{(8)}, 0_{(8)}, Y_{(8)}], [0_{(8)}, Y_{(8)}, 0_{(8)}, 0_{(8)}], [Y_{(8)}, 0_{(8)}, 0_{(8)}, 0_{(8)}]$
$[0_{(8)}, X_{(8)}, 0_{(8)}, 0_{(8)}]$	$[0_{(8)}, 0_{(8)}, 0_{(8)}, Y_{(8)}], [0_{(8)}, 0_{(8)}, Y_{(8)}, 0_{(8)}], [Y_{(8)}, 0_{(8)}, 0_{(8)}, 0_{(8)}]$
$[X_{(8)}, 0_{(8)}, 0_{(8)}, 0_{(8)}]$	$[0_{(8)}, 0_{(8)}, 0_{(8)}, Y_{(8)}], [0_{(8)}, 0_{(8)}, Y_{(8)}, 0_{(8)}], [0_{(8)}, Y_{(8)}, 0_{(8)}, 0_{(8)}]$

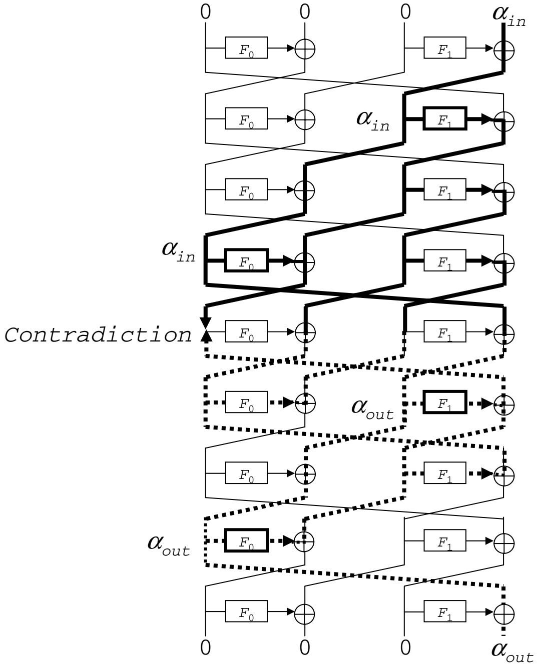


Fig. 3. Nine-round impossible differential

Proof. Assume that the input difference $\Delta x^{\{4,0\}}$ of the fifth-round F_0 function for when the input difference is $[0_{(32)}, 0_{(32)}, 0_{(32)}, [0_{(8)}, 0_{(8)}, 0_{(8)}, X_{(8)}]]$ and the input difference $\Delta x'^{\{4,0\}}$ of the fifth-round F_0 function for when the output difference is $[0_{(32)}, 0_{(32)}, 0_{(32)}, [Y_{(8)}, 0_{(8)}, 0_{(8)}, 0_{(8)}]]$ are the same.

$$\Delta x^{\{4,0\}} = \Delta x'^{\{4,0\}}. \tag{1}$$

The $\Delta x^{\{4,0\}}$ can be expressed using the fourth-round matrix M_0 and second-round matrix M_1 as

$$\begin{aligned} \Delta x^{\{4,0\}} &= M_0 \, {}^t[0, 0, 0, X'] \oplus M_1 \, {}^t[0, 0, 0, X''] \\ &= (M_0 \,|\, M_1) \, {}^t[0, 0, 0, X', 0, 0, 0, X''], \end{aligned} \tag{2}$$

where X' is the output difference for when the S_1 input difference is X, and X'' is the output difference for when the S_0 input difference is X; both are non-zero values.

Also, the $\Delta x'^{\{4,0\}}$ can be expressed using the 8th-round matrix M_0 and the 6th-round matrix M_1 as

$$\begin{aligned}\Delta x'^{\{4,0\}} &= M_0\,{}^t[Y',0,0,0] \oplus M_1\,{}^t[Y'',0,0,0]\\ &= (M_0\,|\,M_1)\,{}^t[Y',0,0,0,Y'',0,0,0],\end{aligned} \tag{3}$$

where Y' is the output difference for when the S_0 input difference is Y and Y'' is the output difference for when the S_1 input difference is Y; both are non-zero values.

From (1), (2) and (3), we obtain

$$(M_0\,|\,M_1)\,{}^t[Y',0,0,X',Y'',0,0,X''] = {}^t[0,0,0,0] \tag{4}$$

because

$$\begin{aligned}&\Delta x^{\{4,0\}} \oplus \Delta' x^{\{4,0\}}\\ &= (M_0\,|\,M_1)\,{}^t[0,0,0,X',0,0,0,X''] \oplus (M_0\,|\,M_1)\,{}^t[Y',0,0,0,Y'',0,0,0]\\ &= (M_0\,|\,M_1)\,{}^t([0,0,0,X',0,0,0,X''] \oplus [Y',0,0,0,Y'',0,0,0])\\ &= (M_0\,|\,M_1)\,{}^t[Y',0,0,X',Y'',0,0,X''].\end{aligned}$$

From the CLEFIA specifications, the branch number of the concatenation matrix $M_0\,|\,M_1$ is 5. Therefore

$$w_b([Y',0,0,X',Y'',0,0,X'']) + w_b((M_0\,|\,M_1)\,{}^t[Y',0,0,X',Y'',0,0,X'']) \geq 5.$$

From $w_b([Y',0,0,X',Y'',0,0,X'']) = 4$, for the left side of (4),

$$w_b((M_0\,|\,M_1)\,{}^t[Y',0,0,X',Y'',0,0,X'']) \geq 1. \tag{5}$$

Furthermore, for the right side of (4),

$$w_b([0,0,0,0]) = 0. \tag{6}$$

Equations (5) and (6) contradict (4).

Accordingly, $\Delta x^{\{4,0\}}$ and $\Delta x'^{\{4,0\}}$ cannot be equal and $[0,0,0,[0,0,0,X]] \not\rightarrow_{9r} [0,0,0,[Y,0,0,0]]$ is thus an impossible differential. □

3.2 Key Recovery Attack on 11-Round CLEFIA

In this section, we explain an impossible differential attack on 11-round CLEFIA using the 9-round impossible differentials presented in Sect. 3.1 as preparation for an impossible differential attack on 12-round CLEFIA which we show in Sect. 3.3. For simplicity of explanation in the next section, we regard the first-round output to be plaintext and present the attack procedure for the 11 rounds from the second round to the 12th round. Of the 9-round impossible differentials shown in Sect. 3.1, we describe the case for the input difference of $[0,0,0,[0,0,0,X]]$ and the output difference of $[0,0,0,[Y,0,0,0]]$ as shown in Fig. 4. It is possible to recover RK_{22}, RK_{23}, and the most significant byte of $WK_2 \oplus RK_{21}$, which we represent as $RK'_{21,0(8)}$.

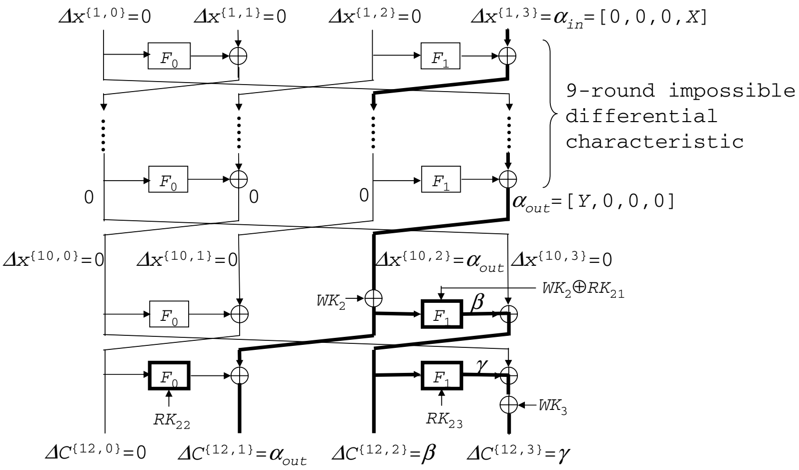


Fig. 4. Impossible differential attack on 11-round CLEFIA

Movement of Whitening Key $\boldsymbol{WK_2}$. Move the whitening key WK_2, and place it at the bit-wise exclusive OR with the 10th-round output $x^{\{10,2\}}$ and bit-wise exclusive OR with RK_{21}. This movement is an equivalent transformation.

Key Recovery. Of the ciphertext pairs that correspond to the plaintext pairs for which the difference is $[0,0,0,[0,0,0,X]]$, choose those for which the ciphertext difference is $[0,[Y,0,0,0],\beta_{(32)},\gamma_{(32)}]$. Here, β represents the 255 values that can be obtained as the output difference when the input difference for M_1 is $[Y,0,0,0]$; γ is a non-zero value. The probability of obtaining such ciphertext pairs is $1/2^{32}\cdot 255/2^{32}\cdot 255/2^{32}\cdot(2^{32}-1)/2^{32}\approx 2^{-80}$.

For the chosen ciphertext pair, all of the keys that are obtained by differential table[1] look-up indexed on the input differences and the output differences of the 11th-round F_1 and the 12th-round F_1 as the key are wrong keys. Those keys are marked as wrong keys in a key table for distinguishing whether $RK'_{21,0}\,|\,RK_{22}\,|\,RK_{23}$ candidates[2] are correct keys or wrong keys. This method is generally used with the objective of finding the correct key by differential attacks; in impossible differential attacks, it can be used to find wrong keys without exhaustive search. The probability of a candidate for $RK'_{21,0}\,|\,RK_{22}\,|\,RK_{23}$ being a wrong key as the result of using two F_1 differential tables is 2^{-40} from the average 2^{-8} probability for the 11th-round F_1 and the average 2^{-32} probability for the 12th-round F_1. Accordingly, the number of ciphertext pairs required to

[1] A table that records the input value pairs for which occur the input-output differences for each of the input differences and output differences.

[2] To calculate the input value of the 11th-round F_1, it is necessary to try all of RK_{22}. It is therefore useful to have the $RK'_{21,0}\,|\,RK_{23}$ key table when guessing RK_{22}, but we chose to add RK_{22} to the key table as well to simplify the explanation of the 12-round attack in Sect. 3.3.

narrow the candidates down to a single 72-bit correct key $RK'_{21,0} \,|\, RK_{22} \,|\, RK_{23}$, N, is about $2^{45.7}$, from

$$2^{72}(1-2^{-40})^N = 1.$$

From the above facts, $2^{45.7}/2^{-80} = 2^{125.7}$ plaintext pairs are required for attack. If we choose two different plaintexts from a set of 2^8 plaintexts (referred to simply as 'structure' below) for which the first three words and the first three bytes of the fourth word of the plaintext are fixed, we can make ${}_{2^8}C_2 \approx 2^{14.9}$ pairs for which the difference is $[0,0,0,[0,0,0,X]]$. In other words, it is possible to obtain the number of ciphertext pairs that are required for the attack by choosing $2^{110.8}$ $(= 2^{125.7-14.9})$ structures. In that case, the number of plaintexts is $2^{110.8} \cdot 2^8 = 2^{118.8}$.

The time complexity for attack is as follows.

1. For obtaining the ciphertexts : 2^{119} encryptions
2. For reducing the key candidates : $2^{46} \cdot 2^{32} = 2^{78}$ F-function computations $< 2^{73}$ encryptions
   (In detail, $2^{45.7}$ ciphertext pairs $\cdot$ 2^{32} RK_{22} guesses)

Accordingly, the time complexity is 2^{119} encryptions.

The memory used for attack is occupied by the key table and the ciphertext table. The size of the key table, if indexed by the key values, is 2^{72} bits. The size of the ciphertext table is 2^8 blocks (128 bits per block), if indexed by the plaintext values. Accordingly, the memory required for attack is about 2^{65} blocks.

3.3 Key Recovery Attack on 12-Round CLEFIA

We extend the impossible differential attack of the 11-round CLEFIA described in Sect. 3.2 by one round on the plaintext side. In addition to RK_{22}, RK_{23}, and $RK'_{21,0}$, we can obtain the least significant byte of RK_0.

Movement of Whitening Key $\boldsymbol{WK_0}$. Move the whitening key WK_0, and place it at the bit-wise exclusive OR with the first round output $x^{\{1,0\}}$.

Plaintext Choice Method. Prepare a data set that comprises 2^{40} plaintexts in which the first three bytes of the first word, and the third and fourth words of the plaintext are fixed as shown in Fig. 5. In other words, there are 2^{40} plaintexts for which the first three bytes of the fourth word $x^{\{1,3\}}$, the second word $x^{\{1,1\}}$, and the third word $x^{\{1,2\}}$ are fixed, if taken as the first-round output. If, for each value of the first word $x^{\{1,0\}}$ of the first-round output, it is possible to choose 2^8 plaintexts for which the least significant bytes of the fourth word $x^{\{1,3\}}$ are different (i.e., structures), the attack described in Sect. 3.2 can be applied.

Let the first word $x^{\{0,0\}}$ of the plaintext be $[a_{(8)}, b_{(8)}, c_{(8)}, d_{(8)}]$ and let RK_0 be $[k_{0(8)}, k_{1(8)}, k_{2(8)}, RK_{0,3(8)}]$. Here, a,b, and c are arbitrary fixed values, and d is a variable that takes values from 0 to 255 in order. Using this variable to express the first word $x^{\{1,0\}}$ of the first-round output, we get

$$\begin{aligned} x^{\{1,0\}} &= {M_0}^t[S_0(a \oplus k_0), S_1(b \oplus k_1), S_0(c \oplus k_2), 0] \\ &\oplus {M_0}^t[0,0,0,S_1(d \oplus RK_{0,3})] \oplus x^{\{0,1\}}. \end{aligned} \tag{7}$$

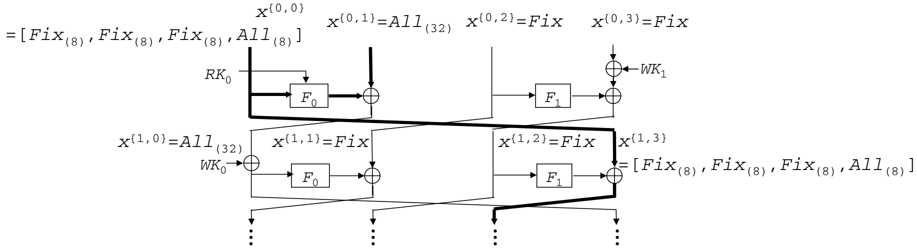


Fig. 5. Choice of plaintext for a one-round extension

The first term on the right side of (7) is a fixed value.

To choose 2^8 plaintexts (structure) such that the least significant bytes of $x^{\{1,3\}}$ are all different for each value of $x^{\{1,0\}}$, we guess $RK_{0,3}$ and choose the data for which $x^{\{0,1\}}$ is $x^{\{1,0\}} \oplus M_0{}^t[0,0,0,S_1(d \oplus RK_{0,3})]$ corresponding to the change in d. Here, $x^{\{1,0\}}$ is actually the unknown value $x^{\{1,0\}} \oplus M_0{}^t[S_0(a \oplus k_0), S_1(b \oplus k_1), S_0(c \oplus k_2), 0]$, but when choosing a single structure, we can fix the value of $x^{\{1,0\}}$. As a result, 2^{32} structures can be chosen for the first-round output.

Key Recovery. Because an attack in the same way as described in Sect. 3.2 is possible, this description follows the procedure of that section.

From among the ciphertext pairs that correspond to the plaintext pairs for which the second-round input difference is $[0,0,0,[0,0,0,X]]$, choose those for which the ciphertext difference is $[0,[Y,0,0,0],\beta,\gamma]$. The probability of obtaining such ciphertext pairs is 2^{-80}.

For the chosen ciphertext pair, the keys for which the 10th-round output difference $[\Delta x^{\{10,0\}}, \Delta x^{\{10,1\}}, \Delta x^{\{10,2\}}, \Delta x^{\{10,3\}}]$ is $[0,0,[Y,0,0,0],0]$ are wrong keys. Prepare a key table to distinguishing whether the $RK'_{21,0} \,|\, RK_{22} \,|\, RK_{23}$ candidate is correct or wrong for each first-round guessed key $RK_{0,3}$. Keys obtained by differential table look-up with the input differences and the output differences for the 11th-round F_1 and the 12th-round F_1 are wrong keys. The probability of a wrong key obtained as an $RK'_{21,0} \,|\, RK_{22} \,|\, RK_{23}$ candidate using the two differential tables is 2^{-40}. Accordingly, the number of ciphertext pairs needed to narrow the 8-bit keys $RK_{0,3}$ and 72-bit keys $RK'_{21,0} \,|\, RK_{22} \,|\, RK_{23}$ down to the correct key, N, is $2^{45.8}$ according to

$$2^{80}(1 - 2^{-40})^N = 1.$$

When key $RK_{0,3}$ is wrong, all of the keys are wrong.

From the above description, $2^{45.8}/2^{-80} = 2^{125.8}$ plaintext pairs are required for attack. Here, by changing the order of choosing the plaintext-ciphertext pairs according to the guessing of key $RK_{0,3}$, the number of chosen plaintexts does not increase when guessing key $RK_{0,3}$. If we choose two different plaintexts from a single structure seen in the first-round output, we can make ${}_{2^8}C_2 \approx 2^{14.9}$

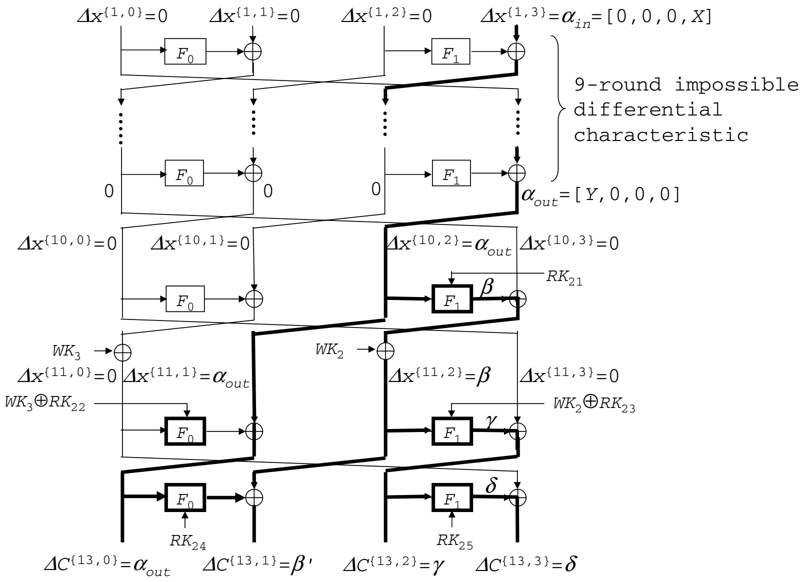


Fig. 6. Impossible differential attack on 13-round CLEFIA

pairs for which the difference is $[0, 0, 0, [0, 0, 0, X]]$. That is to say, if we prepare $2^{78.9}$ $(= 2^{125.8-32-14.9})$ sets of 2^{40} plaintexts (2^{32} structures) for which the first three bytes of the first word and the third and fourth words of the plaintext are fixed, we can obtain the number of ciphertext pairs required for attack. The number of plaintexts in that case is $2^{78.9} \cdot 2^{40} = 2^{118.9}$. The difference in the required number of plaintexts with Sect. 3.2 ($2^{118.8}$) arises from the difference in the number of ciphertext pairs N required to narrow down the keys to the one correct remaining key using the key table.

The time complexity required for attack is as follows.

1. For obtaining the ciphertexts : 2^{119} encryptions
2. For reducing the key candidates : $2^8 \cdot 2^{46} \cdot 2^{32} = 2^{86}$ F-function computations $< 2^{82}$ encryptions
   (In detail, $RK_{0,3}$ guesses $2^8 \cdot$ ciphertext pairs $2^{45.8} \cdot RK_{22}$ guesses 2^{32})

Accordingly, the time complexity is 2^{119} encryptions.

The memory used for attack is occupied by the key table and the ciphertext table. The key table size is 2^{80} bits and the ciphertext table size is 2^{40} blocks. Accordingly, the memory size required for attack is about 2^{73} blocks.

3.4 Key Recovery Attacks on 13 and 14-Round CLEFIA

We present a 13-round CLEFIA attack for the key length of 192 bits or more shown in Fig. 6 and a 14-round CLEFIA attack for the key length of 256 bits.

In the 13-round attack, it is possible to obtain $RK_{0,3}$, the most significant byte of RK_{21} (denoted as $RK_{21,0(8)}$), $WK_3 \oplus RK_{22}$, $WK_2 \oplus RK_{23}$, RK_{24} and

RK_{25}. In the 14-round attack, it is possible to obtain $RK_{0,3}$, the most significant byte of $WK_3 \oplus RK_{21}$, RK_{22}, RK_{23}, $WK_3 \oplus RK_{24}$ and $WK_2 \oplus RK_{25}$, RK_{26}, and RK_{27}. In the same way as done in Sects. 3.2 and 3.3, we first present the attack procedure for the 12 rounds from the second round to the 13th round. Then, we extend one round on the plaintext side. Finally, we explain the 14-round attack.

Movement of Whitening keys $\boldsymbol{WK_0}$, $\boldsymbol{WK_2}$, and $\boldsymbol{WK_3}$. The whitening keys WK_0, WK_2, WK_3 are moved in the same way as in Sects. 3.2 and 3.3.

Key Recovery on 12-Round CLEFIA. We choose the ciphertext pairs for which the first round output difference is $[0, 0, 0, [0, 0, 0, X]]$ and the 12th-round output difference is $[[Y, 0, 0, 0], \beta, \gamma, 0]$ for use in attack. Here, β represents the 255 values that can be obtained as the output difference when the input difference for M_1 is $[Y, 0, 0, 0]$; γ is a non-zero value.

From among the ciphertext pairs that correspond to the plaintext pairs for which the first round output difference is $[0, 0, 0, [0, 0, 0, X]]$, select those for which the differences are $[[Y, 0, 0, 0], \beta'_{(32)}, \gamma, \delta_{(32)}]$. Here, β' is the bit-wise exclusive OR of the 255 values that β can take with the 255 values that the M_0 output difference can take for the case in which the input difference of M_0 is $[Y, 0, 0, 0]$, or $255 \cdot 255 \approx 2^{16}$. The γ and δ are non-zero values. The probability of obtaining such ciphertext pairs is

$$255/2^{32} \cdot (255 \cdot 255)/2^{32} \cdot (2^{32} - 1)/2^{32} \cdot (2^{32} - 1)/2^{32} \approx 2^{-40}.$$

From among the chosen ciphertext pairs, classify the ciphertext pairs for which the 12th-round output difference is $[[Y, 0, 0, 0], \beta, \gamma, 0]$ by guessing the most significant byte of RK_{24}. Among the ciphertext pairs for which the difference is $[[Y, 0, 0, 0], \beta', \gamma, \delta]$, the probability that a usable ciphertext pair exists for each value of the most significant byte RK_{24} is 2^{-8}.

The keys for which the 10th-round output difference is $[0, 0, [Y, 0, 0, 0], 0]$ are wrong keys. Prepare a table (key table) for distinguishing $RK_{21,0} \,|\, (WK_2 \oplus RK_{23}) \,|\, RK_{25}$ candidates as correct or wrong. Then, use the input differences and output differences for the 11th-round and 12th-round F_1s and the 13th-round F_1 for look-up in the differential table and mark the obtained keys as wrong. Here, to calculate the input values of the 11th-round F_1 and the 12th-round F_1, we guess the least significant three bytes of RK_{24} and all of $WK_3 \oplus RK_{22}$. The input of the 12th-round F_0 can be calculated using the RK_{25} candidates.

The probability of knowing that a $RK_{21,0} \,|\, (WK_2 \oplus RK_{23}) \,|\, RK_{25}$ candidate is wrong by using the differential table for the three F_1s is 2^{-72}, from the average of 2^{-8} for the 11th-round F_1 and the average of 2^{-32} for the 12th-round and 13th-round F_1. Accordingly, the number of ciphertext pairs, N, required to narrow the 72-bit key $RK_{21,0} \,|\, (WK_2 \oplus RK_{23}) \,|\, RK_{25}$ and 64-bit key $RK_{24} \,|\, (WK_3 \oplus RK_{22})$ down to the correct key is about $2^{78.6}$ from

$$2^{136}(1 - 2^{-72})^N = 1.$$

From the above description, the number of plaintext pairs required for attack is $2^{78.6-40-8} = 2^{126.6}$.

If we choose two plaintexts from the same structure, we can make ${}_{2^8}C_2 \approx 2^{14.9}$ pairs for which the difference is $[0, 0, 0, [0, 0, 0, X]]$. That is to say, if we choose $2^{111.7}(= 2^{126.6-14.9})$ structures, we can obtain the number of ciphertext pairs required for attack. In that case, the number of plaintexts is $2^{111.7} \cdot 2^8 = 2^{119.7}$.

Key Recovery on 13-Round CLEFIA. We extend the method for attack the 12-round CLEFIA that is described above by one round on the plaintext side to break 13-round CLEFIA.

The number of ciphertext pairs, N, required to narrow down the 8-bit key $RK_{0,3}$, the 72-bit key $RK_{21,0} \mid (WK_2 \oplus RK_{23}) \mid RK_{25}$ and the 64-bit key $RK_{24} \mid (WK_3 \oplus RK_{22})$ to the one correct key using the key table is $2^{78.7}$ according to

$$2^{144}(1 - 2^{-72})^N = 1.$$

The method for choosing structures for each value of the first word $x^{\{1,0\}}$ of the first round output is the same as described in Sect. 3.3, so the number of chosen plaintexts on the plaintext side is extended by N. Accordingly, the number of plaintexts required is $2^{119.8}$.

Prepare $2^{79.8}$ sets of 2^{40} plaintexts for which the first three bytes of the first word and the third and fourth words are fixed ($2^{119.8}$ plaintexts in total). Regarding these plaintexts at the first round output, we can consider them to be $2^{79.8}$ sets of 2^{40} plaintexts with the first three bytes of the fourth word and second and third words fixed. We save these $2^{119.8}$ plaintexts in a table, guess $RK_{0,3}$, and choose the plaintext pairs and use them in attack. The reason for saving all of the data, which differs from the procedure of Sect. 3.3, is that there are more keys to be guessed on the ciphertext side, and it is not possible to have a key table for them.[3]

The time complexity required for attack is as follows.

1. For obtaining the ciphertexts : $2^{119.8}$ encryptions
2. For reducing the key candidates : $2^8 \cdot 2^{78.7} \cdot 2^{64} = 2^{150.7}$ F-function computations $< 2^{146}$ encryptions
   (In detail, 2^8 $RK_{0,3}$ guesses $\cdot$ $2^{78.7}$ ciphertext pairs $\cdot$ 2^{64} $WK3 \oplus RK_{22}$ and RK_{24} guesses)

Accordingly, the time complexity is 2^{146} encryptions.

The memory used for attack is occupied by the key table and the ciphertext table. The size of the key table is 2^{72} bits; the size of the ciphertext table is $2^{119.8}$ blocks. Accordingly, the memory required for attack is about 2^{120} blocks.

Key Recovery on 14-Round CLEFIA. 14-round CLEFIA can be broken by adding exhaustive search of the 14th-round keys RK_{26} and RK_{27} to the 13-round attack. The number of chosen plaintexts required for attack is $2^{120.3}$, because the number of ciphertext pairs, N, required for narrowing the keys down to the correct key using the key table is about $2^{79.2}$, from

$$2^{208}(1 - 2^{-72})^N = 1.$$

[3] In this paper, it is not possible to have a table that exceeds 2^{128} blocks.

The time complexity is as follows.

1. For obtaining the ciphertexts : $2^{120.3}$ encryptions
2. For reducing the key candidates : $2^8 \cdot 2^{79.2} \cdot 2 \cdot 2^{128} = 2^{216.2}$ F-function computations $< 2^{212}$ encryptions
   (In detail, $2^8 RK_{0,3}$ guesses $\cdot 2^{79.2}$ ciphertext pairs $\cdot\ 2^{128}$ guesses for RK_{22}, $WK3 \oplus RK_{26}$ and RK_{27} guesses)

Accordingly, the time complexity is 2^{212} encryptions.

The memory used for attack is occupied by the key table and the ciphertext table. The size of the key table is 2^{72} bits; the size of the ciphertext table is $2^{120.3}$ blocks. Accordingly, the amount of memory required for attack is about 2^{121} blocks.

4 Conclusion

We have presented previously unknown 9-round impossible differentials in CLEFIA, which are impossible differentials that exist in structures designed by using DSM. We used these impossible differentials to apply impossible differential attacks on CLEFIA. The result showed that an impossible differential attack that is more efficient than exhaustive search is possible for 128-bit key, 12-round CLEFIA. Furthermore, attack of 13-round CLEFIA and 14-round CLEFIA is possible for key lengths of 192 bits and 256 bits, respectively. The number of chosen plaintexts, the time complexity, and the amount of memory required for attack are listed in Table 2.

Table 2. Results of impossible differential attacks

Reference	Number of rounds	Key length	Chosen plaintexts	Time complexity (encryptions)	Amount of memory (blocks)
[6,7]	10	128, 192, 256	$2^{101.7}$	2^{102}	2^{32}
[6,7]	11	192, 256	$2^{103.5}$	2^{188}	2^{121}
[6,7]	12*	256	$2^{103.8}$	2^{252}	2^{153}
This paper	12	128, 192, 256	$2^{118.9}$	2^{119}	2^{73}
This paper	13	192, 256	$2^{119.8}$	2^{146}	2^{120}
This paper	14	256	$2^{120.3}$	2^{212}	2^{121}

* Without whitening key

Even though the 9-round impossible differentials presented in this paper have the same number of rounds characteristic as the impossible differentials identified by the designers, our impossible differential attacks exceed the designers' evaluation by two more rounds that can be broken for each key length. That is true for the following reason. For the impossible differentials found by the designers, the length of the parts of the plaintext differences and ciphertext differences that are not zero is 32 bits, and the plaintext differences and ciphertext differences must be the same. For our impossible differentials, however, the length of those parts is 8 bits, and it is not necessary for the plaintext differences and the ciphertext differences to be the same, that is, they are truncated differences. If the number of bits for which the difference is non-zero is small, the number of round key bits related to the difference is also small. Because it is possible to obtain round keys that span many rounds, the number of rounds that can be broken can be increased. Also, because it is a truncated difference, the probability of obtaining ciphertext that can be used in attack is high, and we were able to increase the number of rounds that can be broken by reducing the number of chosen plaintexts that are required. Other reasons for the successful attack are the movement of the whitening key and the use of the differential table method that is often used in differential attacks. Because the number of CLEFIA rounds is 18 for a key length of 128 bits, 22 for a 192-bit key and 26 for a 256-bit key, the impossible differential attacks presented in this paper do not affect the security of CLEFIA. These attacks can, however, break more rounds of than other CLEFIA attack methods.

There is currently no method for guaranteeing resistance to an impossible differential attack and no method for designing a block cipher that is resistant to an impossible differential attack. Accordingly, much time should be allocated to evaluation of block cipher with respect to impossible differential attacks. Furthermore, methods for guaranteeing resistance to an impossible differential attack and methods for designing block ciphers that resist impossible differential attacks are important topics for future research.

Acknowledgments. The authors would like to thank Takeshi Kawabata, Hiroki Nakashima, Takahiko Syouji, and Akira Nozawa for their helpful comments.

References

1. Biham, E., Biryukov, A., Shamir, A.: Cryptanalysis of Skipjack Reduced to 31 Rounds Using Impossible Differentials. In: Stern, J. (ed.) EUROCRYPT 1999. LNCS, vol. 1592, pp. 12–23. Springer, Heidelberg (1999)
2. Biham, E., Shamir, A.: Differential Cryptanalysis of DES-like Cryptosystems. In: Menezes, A. J., Vanstone, S.A. (eds.) CRYPTO 1990. LNCS, vol. 537, pp. 2–21. Springer, Heidelberg (1991)
3. Matsui, M.: Linear Cryptanalysis Method for DES Cipher. In: Helleseth, T. (ed.) EUROCRYPT 1993. LNCS, vol. 765, pp. 386–397. Springer, Heidelberg (1994)
4. Shirai, T., Preneel, B.: On Feistel Ciphers Using Optimal Diffusion Mappings Across Multiple Rounds. In: Lee, P.J. (ed.) ASIACRYPT 2004. LNCS, vol. 3329, pp. 1–15. Springer, Heidelberg (2004)
5. Shirai, T., Shibutani, K.: On Feistel Structures Using a Diffusion Switching Mechanism. In: Robshaw, M.J.B. (ed.) FSE 2006. LNCS, vol. 4047, pp. 41–56. Springer, Heidelberg (2006)
6. Shirai, T., Shibutani, K., Akishita, T., Moriai, S., Iwata, T.: The 128-bit Blockcipher CLEFIA (Extended Abstract). In: Biryukov, A. (ed.) FSE 2007. LNCS, vol. 4593, pp. 181–195. Springer, Heidelberg (2007)
7. Sony Corporation. The 128-bit Blockcipher CLEFIA, Security and Performance Evaluations, Revision 1.0, June 1 (2007), http://www.sony.co.jp/Products/clefia/

MD4 is Not One-Way

Gaëtan Leurent

École Normale Supérieure – Département d'Informatique,
45 rue d'Ulm, 75230 Paris Cedex 05, France
Gaetan.Leurent@ens.fr

Abstract. MD4 is a hash function introduced by Rivest in 1990. It is still used in some contexts, and the most commonly used hash functions (MD5, SHA-1, SHA-2) are based on the design principles of MD4. MD4 has been extensively studied and very efficient collision attacks are known, but it is still believed to be a one-way function.

In this paper we show a partial pseudo-preimage attack on the compression function of MD4, using some ideas from previous cryptanalysis of MD4. We can choose 64 bits of the output for the cost of 2^{32} compression function computations (the remaining bits are randomly chosen by the preimage algorithm).

This gives a preimage attack on the compression function of MD4 with complexity 2^{96}, and we extend it to an attack on the full MD4 with complexity 2^{102}. As far as we know this is the first preimage attack on a member of the MD4 family.

Keywords: MD4, hash function, cryptanalysis, preimage, one-way.

1 Introduction

Hash functions are fundamental cryptographic primitives used in many constructions and protocols. A hash function takes a bitstring of arbitrary length as input, and outputs a digest, a small bitstring of fixed length n.

$$F : \{0,1\}^* \mapsto \{0,1\}^n$$

When used in a cryptographic context, we expect a hash function to behave somewhat like a random oracle. The digest is used as a kind of fingerprint: it can be used to test whether two documents are equal, but should neither reveal any other information about the input nor be malleable. More concretely, we ask a cryptographic hash function to resist three major attacks:

Collision: Given F, find $M_1 \neq M_2$ s.t. $F(M_1) = F(M_2)$.
Second-preimage: Given F and M_1, find $M_2 \neq M_1$ s.t. $F(M_1) = F(M_2)$.
Preimage: Given F and $\overline{H}$, find M s.t. $F(M) = \overline{H}$.

Due to the birthday paradox, we have a generic collision attack with complexity $2^{n/2}$, while brute force preimage or second-preimage attacks have complexity 2^n: this defines the security requirements of a n-bit hash function. Collision

K. Nyberg (Ed.): FSE 2008, LNCS 5086, pp. 412–428, 2008.

resistance is the strongest notion, so most constructions use a collision resistant hash function, and most cryptanalysis target collision attack. For a more formal definition of these properties, and the relations between them, see [18,17].

Unfortunately, many currently used hash functions have been broken by collision attacks: MD4 [5,21,19] (the best attack has complexity 2^1), MD5 [23,10] (best attack: 2^{23}), and SHA-1 [22,13] (best attack: 2^{60}). These functions are now considered unsafe but in practice very few constructions or protocols are really affected.

In this paper we consider preimage resistance, which is a weaker security notion and is still believed to hold for these hash functions. In particular MD4 is broken by collision attacks since 1996 but it is still used in some applications where speed is important and/or a one-way function is needed but collision resistance is not important:

- to "encrypt" passwords in Windows NT and later (as the NTLM hash);
- for password derivation in the S/KEY one time password system [8];
- to compare file blocks in the incremental file transfer program rsync;
- for file identification and integrity in the eDonkey peer-to-peer network.

S/Key and rsync even use a truncated MD4 and rely on the partial one-wayness of MD4.

Preimage attacks are rather rare in the world of hash function cryptanalysis; the most notable example is the preimage attack against MD2 by Muller [15], later improved by Knudsen and Mathiassen [11] which has a complexity of 2^{97}. A preimage attack has much more impact than a collision attack: it can be used to fool integrity checks, to forge signatures using only known messages, to break "encrypted" password files,... Moreover, when the hash function follows the Merkle-Damgård paradigm (this is the case for MD4) we can add any chosen prefix: given a message M and a target hash value $\overline{H}$, we can actually compute N such that $\text{MD4}(M||N) = \overline{H}$. For instance, this can be used to create a malicious software package with a given signature when trailing garbage is allowed (*eg.* this is the case with zip, gzip, and bzip2 files).

1.1 Our Results

Our main result is a preimage attack against MD4 with complexity 2^{102}. This attack uses messages of 18 blocks or slightly more (more precisely 9151 bits, about one kilobyte), and we can add any chosen prefix.

This is based on a partial pseudo-preimage attack on the compression function: we can choose 64 bits of the output (the other bits being randomly chosen by the preimage algorithm) and 32 bits of the input for the cost of 2^{32} compression function (brute force would require 2^{64}).

Our attack uses many ideas from previous cryptanalysis of MD4 [5,20,6,21]. We consider MD4 as a system of equation, we use some kind of differential path and use the Boolean functions to absorb some differences, we fix many values of the internal state using some particularities of the message expansion.

1.2 Related Work

Md4 has been introduced as a cryptographic hash function by Rivest [16], in 1990 and many cryptanalytic effort has been devoted to study its security. The design principles of MD4 are used in MD5 and the SHA family, which are the most widely used hash function today. Any result about MD4 is interesting by itself, and also gives some insight to the security level of the other members of the MD4 family.

Shortly after the introduction of MD4, collision attacks were found on reduced variants of MD4: den Boer and Bosselaers [4] found an attack against the last two rounds, and Merkle had an unpublished attack against the first two rounds. Another attack against the first two rounds was later found by Vaudenay [20]. The first collision attack against the full MD4 is due to Dobbertin [5] in 1996. More recently, Wang *et. al.* found a very efficient collision attack on MD4 [21], which was later improved by Sasaki *et. al.* [19] and only costs 2 compression functions. Due to all these attacks MD4 is no longer used as a collision-resistant hash function.

The main result concerning the one-wayness of MD4 is due to Dobbertin [6]. He showed that if the last round of MD4 is removed, preimages can be found in the resulting hash function with a complexity of 2^{32} compression function calls. This work was studied and improved using SAT solvers by De *et al.* [2]. They managed to invert up to 2 rounds and 7 steps of MD4. To the best of our knowledge, no preimage attack has been found on the full MD4 with three rounds.

Recently, Yu *et al.* found a kind of second-preimage attack on MD4 [24]. However this kind of attack is not what we usually call a second-preimage attack because it only works for a small subset of the message space. This attack has a complexity of one compression function, but it works only with probability 2^{-56} and cannot be repeated when it fails. If we want to build an attack that works for any message out of this, we will use a brute-force search when the attack fails: it will have a workload of 1 with probability 2^{-56}, and a workload of $1+2^{128}$ with probability $1-2^{-56}$; the average workload is still extremely close to 2^{128}. More interestingly, we can use this with long messages: if the message is made of 2^{63} blocks (there is not limitation to the size of the message in MD4, as opposed to SHA-1), we will be able to find a second-preimage for at least one of the blocks with a probability of $1-\exp(-2^{63-56}) \approx 1-2^{-184}$. Thus, the cases where we have to run a brute-force search become negligible, and the average workload is just the time needed to test each block until a good one is found, so we expect it to be 2^{56}.

Another related work due to Kelsey and Schneier [9] introduced a generic second-preimage attack against iterated hash functions using long messages. This is a nice result showing the limitations of the Merkle-Damgård paradigm, but an attack on messages of 2^{64} bits is not really practical. Our attack typically uses messages of 20 blocks (about 1 kilobyte in total).

1.3 Description of MD4

MD4 is an iterated hash function following the Merkle-Damgård paradigm. The message is padded and cut into blocks of k bits (with $k = 512$ for MD4), and the digest is computed by iterating a compression function cF, starting with an initial value IV.

$$cF : \{0,1\}^{n+k} \mapsto \{0,1\}^n$$
$$h_0 = IV, \qquad h_{i+1} = cF(h_i, M_i)$$
$$F(M_0, M_1, ...M_{p-1}) = h_p$$

The padding of MD4 uses the MD strengthening: it is designed to be invertible, and includes the size of the message. The message is first padded with a single `1` bit followed by a variable number of `0`'s, so that the size of the message is congruent to 448 modulo 512. This first step adds between 1 and 512 bits to the message. Then the last 64 bits are filled with the size of the original message modulo 2^{64}. Note that MD4 can hash any bitstring: it is not restricted to hash bytes, and there is no limit to the size of the message.

We will use the following definitions for attacks on the compression function cF:

Pseudo-Preimage: Given cF and $\overline{H}$, find IV, M s.t. $cF(IV, M) = \overline{H}$.
Preimage: Given cF, IV and $\overline{H}$, find M s.t. $cF(IV, M) = \overline{H}$.

The compression function of MD4 is an unbalanced Feistel ladder with an internal state of four 32-bit registers. It is made of 48 steps, where each step updates one of these registers. The 48 steps are divided into 3 rounds of 16 steps; each round reads the 16 message words in a different order (this is a very simple message expansion). To better describe our attack, we will assign the name Q_i to the value computed in the step i: we now have 48 internal state variables, and each one is computed from the 4 preceding ones (we use Q_{-4} to Q_{-1} to denote the IV):

$$\begin{array}{r l} \text{Step update:} & Q_i = (Q_{i-4} \boxplus \varPhi_i(Q_{i-1}, Q_{i-2}, Q_{i-3}) \boxplus m_{\pi(i)} \boxplus k_i) \lll s_i \\ \text{Input:} & Q_{-4} \;\|\; Q_{-1} \;\|\; Q_{-2} \;\|\; Q_{-3} \\ \text{Output:} & Q_{-4} \boxplus Q_{44} \,\|\, Q_{-1} \boxplus Q_{47} \,\|\, Q_{-2} \boxplus Q_{46} \,\|\, Q_{-3} \boxplus Q_{45} \end{array}$$

First round:	$0 \le i < 16$	$\varPhi_i = \text{IF}$	$k_i = K_0 = 0$
Second round:	$16 \le i < 32$	$\varPhi_i = \text{MAJ}$	$k_i = K_1 =$ `0x5a827999`
Third round:	$32 \le i < 48$	$\varPhi_i = \text{XOR}$	$k_i = K_2 =$ `0x6ed9eba1`

π(0...15):	0,	1,	2,	3,	4,	5,	6,	7,	8,	9,	10,	11,	12,	13,	14,	15
π(16...31):	0,	4,	8,	12,	1,	5,	9,	13,	2,	6,	10,	14,	3,	7,	11,	15
π(32...47):	0,	8,	4,	12,	2,	10,	6,	14,	1,	9,	5,	13,	3,	11,	7,	15

We will use the fact that we can fix some values of the internal state instead of only fixing values of the message words: since the step update function is invertible, when Q_{i-1}, Q_{i-2} and Q_{i-3} are known, we can compute any one of Q_i, Q_{i-4} or m_i from the two others (see Algorithm 1 for explicit formulas).

we consider MD4 as a big system of equations over the variables Q_{-4} to Q_{47} and m_0 to m_{15} (we consider only words, and never look at individual bits); we have 48 step update equations, and 4 equations for the output value. We consider the input chaining variables IV_0, IV_1, IV_2, IV_3 as free: this makes the attack on the compression function easier (it's a pseudo-preimage attack), but we will have some work to do to turn it into a preimage attack on the full hash function. We

Algorithm 1. Step functions

```
function MD4StepForward($i$)
    $Q_i \leftarrow (Q_{i-4} \boxplus \Phi_i(Q_{i-1}, Q_{i-2}, Q_{i-3}) \boxplus m_{\pi(i)} \boxplus k_i) \lll s_i$
end function
function MD4StepBackward($i$)
    $Q_{i-4} \leftarrow (Q_i \ggg s_i) \boxminus \Phi_i(Q_{i-1}, Q_{i-2}, Q_{i-3}) \boxminus m_{\pi(i)} \boxminus k_i$
end function
function MD4StepMessage($i$)
    $m_{\pi(i)} \leftarrow (Q_i \ggg s_i) \boxminus Q_{i-4} \boxminus \Phi_i(Q_{i-1}, Q_{i-2}, Q_{i-3}) \boxminus k_i$
end function
```

use $\overline{X}$ to denote the desired value of a variable X, which is given as an input to our attack. The system we are trying to solve can be written as:

$$\begin{cases} Q_i = (Q_{i-4} \boxplus \Phi_i(Q_{i-1}, Q_{i-2}, Q_{i-3}) \boxplus m_{\pi(i)} \boxplus k_i) \lll s_i \quad \text{for } i \in \{0...47\} \\ H_0 = Q_{-4} \boxplus Q_{44} = \overline{H}_0 \\ H_1 = Q_{-3} \boxplus Q_{45} = \overline{H}_1 \\ H_2 = Q_{-2} \boxplus Q_{46} = \overline{H}_2 \\ H_3 = Q_{-1} \boxplus Q_{47} = \overline{H}_3 \end{cases}$$

To make the equations more readable, we will use a grey background to show the variables whose value has already been chosen in a previous step of the algorithm.

We will use $x^{[k]}$ to represent the $(k+1)$-th bit of x, that is $x^{[k]} = (x \ggg k) \bmod 2$ (note that we count bits and steps starting from 0). We also use $\mathbb{0}$ and $\mathbb{1}$ to denote the 32-bit words whose bits are all zeroes and all ones (*i.e.* $\mathbb{0} =$ `0x00000000` and $\mathbb{1} =$ `0xffffffff`).

1.4 Road Map

First we will describe our attack against the compression function cMD4. Then we will show how to extend it to an attack against the full MD4, with a better complexity than the generic attack.

2 Pseudo-preimage on the Compression Function

The general idea of the attack comes from a simple observation: we know that MD4 is very sensitive to differential attacks — there is a collision attack that costs less than 2 calls to the compression function [19]. However, in a preimage attack, we are looking for a single message and differential tools seem unsuitable for this task. Our idea is to start with an *initial message* (IV, M) with very specific properties, and the digest cMD4(IV, M) of this message. Now we will use differential paths to create a family of *related messages* $(IV^{(i)}, M^{(i)})$ so that the computation of cMD4$(IV^{(i)}, M^{(i)})$ differs from the computation of cMD4(IV, M) in a controlled way. This will allow us to choose a particular $(IV^{(i_0)}, M^{(i_0)})$ so that some bits of cMD4($IV^{(i_0)}, M^{(i_0)}$) agree with a target value. Alternatively, we could try to mount a second preimage attack using differential tools, as done in [24], but this will probably only work for a small fraction of the message space, because differential paths *à la* Wang impose many constraints on the message.

Following this idea, we look for a set of constraints on the initial message and a way to derive the related messages. We managed to easily select a related message that agrees with 32 bits of the target hash. We can compute 2^{32} good related messages for a cost of 2^{32} compression function calls: our attack has an amortized cost of 1. When we run it 2^{32} times, we will test 32 more bits of the target hash, and we expect to find a partial pseudo-preimage on 64 bits. Similarly, to find a full pseudo-preimage we expect to repeat it 2^{96} times.

2.1 The Initial Message

The key part is to choose a set of constraints that allow to place many differential paths in MD4, in order to have as many related messages as possible. Instead of looking at single bits, we will consider the 32-bit words of MD4, and try to have 32 paths at once. We will use some properties of the Boolean functions in MD4, and some properties of the message expansion.

The Boolean functions used in the first and second round of MD4 have the nice property to be able to absorb some difference. This key property was used in early cryptanalysis of MD4 [20,6] and is the starting point of the construction of differential paths [7] *à la* Wang. Notably, we have the following properties, where $\mathbb{C}$ is a constant, and x is variable:

Absorb 1st input	Absorb 2nd input	Absorb 3rd input
$\mathrm{IF}(x,\mathbb{C},\mathbb{C})=\mathbb{C}$ $\mathrm{MAJ}(x,\mathbb{C},\mathbb{C})=\mathbb{C}$	$\mathrm{IF}(\mathbb{0},x,\mathbb{C})=\mathbb{C}$ $\mathrm{MAJ}(\mathbb{C},x,\mathbb{C})=\mathbb{C}$	$\mathrm{IF}(\mathbb{1},\mathbb{C},x)=\mathbb{C}$ $\mathrm{MAJ}(\mathbb{C},\mathbb{C},x)=\mathbb{C}$

This can be used to let one message word be free in the first round or in the second round, as shown by Figure 1. If we fix some internal state variables, a change of m_i will only change Q_i, Q_{i+4}, ... Q_{i+4k} and can be corrected by a change in m_{i+4k}. Additionally, we are free to change m_{i+4},m_{i+8},... $m_{i+4(k-1)}$ without breaking any extra internal state.

To choose the step i_0 where we introduce the difference, and i_1 where we cancel it, we will look at the message expansion. We want i_0 and i_1 to be quite far away so as to skip a big part of the compression function and to maximize the number of free message words in between, but we do not want to use the same message word twice between i_0 and i_1. This leave us with 3 possibilities:

- $(i_0, i_1) = (\ 0, 16)$: m_0 , m_4 , m_8 and m_{12} are free
- $(i_0, i_1) = (15, 31)$: m_{15}, m_{12}, m_{13} and m_{14} are free
- $(i_0, i_1) = (16, 32)$: m_0 , m_1 , m_2 and m_3 are free

Note that Vaudenay [20] and Dobbertin [6] used the same idea in their attacks, with $(i_0, i_1) = (15, 31)$. Here we choose $(i_0, i_1) = (16, 32)$ because the free message words are used in the very beginning of the first round. To fix the first

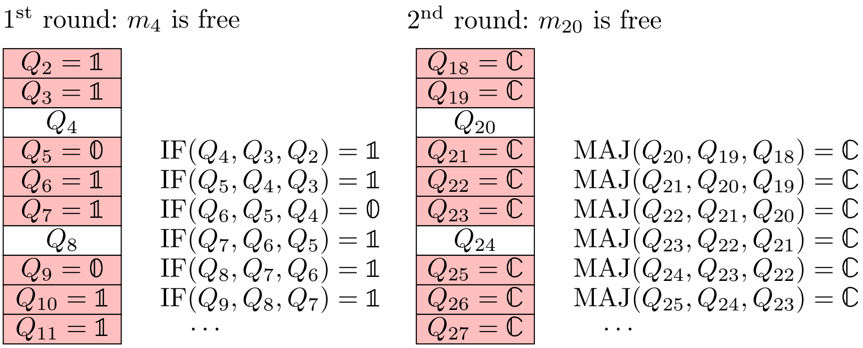
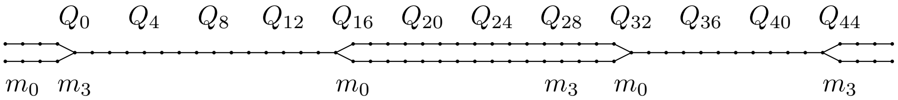

1st round: m_4 is free
$Q_2 = \mathbb{1}$
$Q_3 = \mathbb{1}$
Q_4
$Q_5 = \mathbb{0}$
$Q_6 = \mathbb{1}$
$Q_7 = \mathbb{1}$
Q_8
$Q_9 = \mathbb{0}$
$Q_{10} = \mathbb{1}$
$Q_{11} = \mathbb{1}$
$\mathrm{IF}(Q_4, Q_3, Q_2) = \mathbb{1}$
$\mathrm{IF}(Q_5, Q_4, Q_3) = \mathbb{1}$
$\mathrm{IF}(Q_6, Q_5, Q_4) = \mathbb{0}$
$\mathrm{IF}(Q_7, Q_6, Q_5) = \mathbb{1}$
$\mathrm{IF}(Q_8, Q_7, Q_6) = \mathbb{1}$
$\mathrm{IF}(Q_9, Q_8, Q_7) = \mathbb{1}$
$\cdots$
2nd round: m_{20} is free
$Q_{18} = \mathbb{C}$
$Q_{19} = \mathbb{C}$
Q_{20}
$Q_{21} = \mathbb{C}$
$Q_{22} = \mathbb{C}$
$Q_{23} = \mathbb{C}$
Q_{24}
$Q_{25} = \mathbb{C}$
$Q_{26} = \mathbb{C}$
$Q_{27} = \mathbb{C}$
$\mathrm{MAJ}(Q_{20}, Q_{19}, Q_{18}) = \mathbb{C}$
$\mathrm{MAJ}(Q_{21}, Q_{20}, Q_{19}) = \mathbb{C}$
$\mathrm{MAJ}(Q_{22}, Q_{21}, Q_{20}) = \mathbb{C}$
$\mathrm{MAJ}(Q_{23}, Q_{22}, Q_{21}) = \mathbb{C}$
$\mathrm{MAJ}(Q_{24}, Q_{23}, Q_{22}) = \mathbb{C}$
$\mathrm{MAJ}(Q_{25}, Q_{24}, Q_{23}) = \mathbb{C}$
$\cdots$

Fig. 1. Absorption of a difference. The step update function is: $Q_i = (Q_{i-4} \boxplus \Phi_i(Q_{i-1}, Q_{i-2}, Q_{i-3}) \boxplus m_{\pi(i)} \boxplus k_i) \lll s_i$.

round, we will correct a modification of the free message words using the IV, and this correction will only involve the first 4 steps of the compression function.

We now have a very good differential path with m_0 and m_3: if we consider the set of 2^{32} pairs that keep Q_{32} constant, their effect on the final hash only involves the first 4 steps and the last 4 steps of MD4. The other free variables will be used to simplify the equations so as to make this path easier to use. Schematically, the differential path looks like this:



We can now choose what will be fixed by the initial message and what will be free for the related messages. The message words m_4 to m_{15} will be fixed by the initial message, while m_0 to m_3 are part of the related message. The internal state variables $Q_{14}, Q_{15}, Q_{17}, Q_{18}, Q_{19}, Q_{21}, ...Q_{30}$ need to be equal and will be part of the initial message. Q_{13} is in the initial message because it is fixed by the step 17, and similarly Q_{31} is fixed by step 31. We will add Q_{12} in the initial message to fix the internal state of the first round. See Figure 2 for a graphical representation.

To select an initial message, we choose random values for $\mathbb{C}$, Q_{12}, Q_{13} and m_{15}; this allows us to compute Q_{31} and m_4 to m_{14} in the second round, and Q_0 to Q_{11} in the first round. Thus we can build 2^{128} different initial messages. Each initial message have 2^{128} related messages (by choosing the value of m_0, m_1, m_2, m_3).

2.2 The Related Messages

When an initial message is fixed, we have to choose m_0, m_1, m_2 and m_3 in a way that will give us some control on the hash value. We will first isolate the third

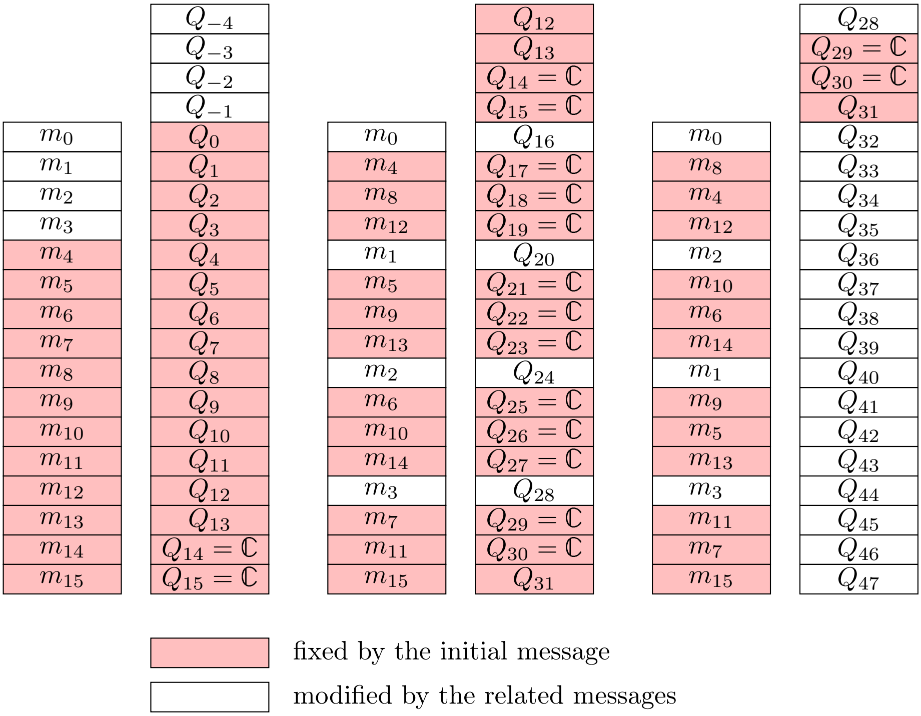

m_0 m_1 m_2 m_3 m_4 m_5 m_6 m_7 m_8 m_9 m_{10} m_{11} m_{12} m_{13} m_{14} m_{15}
Q_{-4} Q_{-3} Q_{-2} Q_{-1} Q_0 Q_1 Q_2 Q_3 Q_4 Q_5 Q_6 Q_7 Q_8 Q_9 Q_{10} Q_{11} Q_{12} Q_{13} $Q_{14} = \mathbb{C}$ $Q_{15} = \mathbb{C}$
m_0 m_4 m_8 m_{12} m_1 m_5 m_9 m_{13} m_2 m_6 m_{10} m_{14} m_3 m_7 m_{11} m_{15}
Q_{12} Q_{13} $Q_{14} = \mathbb{C}$ $Q_{15} = \mathbb{C}$ Q_{16} $Q_{17} = \mathbb{C}$ $Q_{18} = \mathbb{C}$ $Q_{19} = \mathbb{C}$ Q_{20} $Q_{21} = \mathbb{C}$ $Q_{22} = \mathbb{C}$ $Q_{23} = \mathbb{C}$ Q_{24} $Q_{25} = \mathbb{C}$ $Q_{26} = \mathbb{C}$ $Q_{27} = \mathbb{C}$ Q_{28} $Q_{29} = \mathbb{C}$ $Q_{30} = \mathbb{C}$ Q_{31}
m_0 m_8 m_4 m_{12} m_2 m_{10} m_6 m_{14} m_1 m_9 m_5 m_{13} m_3 m_{11} m_7 m_{15}
Q_{28} $Q_{29} = \mathbb{C}$ $Q_{30} = \mathbb{C}$ Q_{31} Q_{32} Q_{33} Q_{34} Q_{35} Q_{36} Q_{37} Q_{38} Q_{39} Q_{40} Q_{41} Q_{42} Q_{43} Q_{44} Q_{45} Q_{46} Q_{47}
fixed by the initial message
modified by the related messages

Fig. 2. Initial message and related message

round from the second round, by choosing the value of Q_{32}. Then the choice of m_2, m_1 and m_3 will give the final state $Q_{44}, ...Q_{47}$, and m_0 will be chosen last: we expect that one value of m_0 will be consistent with the choice of Q_{32}. Note that since m_0 is used in step 0, we can compute H_1, H_2 and H_3 without knowing m_0. Thus, we can choose a good value of m_2, m_1 and m_3 by looking only at the last round and the first 4 steps of the first round, and compute m_0 later in order to correct the second round.

We now study the first steps and the last steps. Our goal is to find efficiently a value of m_2, m_1, and m_3 that solves some of the equations. The message words m_2 and m_1 are used quite far for the last steps, so it will be hard to study how they affect the final state $Q_{44}, ...Q_{47}$: we will only use m_3 to control the hash, while m_2 and m_1 will be used to simplify the equations.

First Steps. We assume that an initial message has been chosen. Let us first study the initial steps of MD4:

$$Q_0 = (Q_{-4} \boxplus \mathrm{IF}(Q_{-1}, Q_{-2}, Q_{-3}) \boxplus m_0) \lll 3 \tag{1}$$

$$Q_1 = (Q_{-3} \boxplus \mathrm{IF}(Q_0, Q_{-1}, Q_{-2}) \boxplus m_1) \lll 7 \tag{2}$$

$$Q_2 = (Q_{-2} \boxplus \mathrm{IF}(Q_1, Q_0, Q_{-1}) \boxplus m_2) \lll 11 \tag{3}$$

$$Q_3 = (Q_{-1} \boxplus \mathrm{IF}(Q_2, Q_1, Q_0) \boxplus m_3) \lll 19 \tag{4}$$

Equation (4) shows that Q_{-1} is completely determined by m_3. Additionally, we will ask that $Q_1 = \mathbb{1}$: this simplify Equation (3) and make Q_{-2} completely determined by m_2, independently of Q_{-1}:

$$Q_2 = (Q_{-2} \boxplus Q_0 \boxplus m_2) \lll 11 \tag{3'}$$

Last Steps. Let us now study the final steps of MD4. We will assume that a value has been chosen for Q_{32}, m_2, m_1: we can now compute Q_{32} to Q_{43} in the third round, and Q_{-2} by equation (3'). This gives us $Q_{46} = \overline{H}_2 - Q_{-2}$.

$$Q_{44} = (Q_{40} \boxplus \mathrm{XOR}(Q_{43}, Q_{42}, Q_{41}) \boxplus m_3 \boxplus K_2) \lll 3 \tag{5}$$
$$Q_{45} = (Q_{41} \boxplus \mathrm{XOR}(Q_{44}, Q_{43}, Q_{42}) \boxplus m_{11} \boxplus K_2) \lll 9 \tag{6}$$
$$Q_{46} = (Q_{42} \boxplus \mathrm{XOR}(Q_{45}, Q_{44}, Q_{43}) \boxplus m_7 \boxplus K_2) \lll 11 \tag{7}$$
$$Q_{47} = (Q_{43} \boxplus \mathrm{XOR}(Q_{46}, Q_{45}, Q_{44}) \boxplus m_{15} \boxplus K_2) \lll 15 \tag{8}$$

Here we see that (7) gives the value $Q_{44} \oplus Q_{45}$. Moreover, we will ask that $Q_{41} \boxplus m_{11} \boxplus K_2 = \mathbb{0}$ so as to simplify (6). We let V be $Q_{42} \oplus Q_{43} \oplus Q_{44} \oplus Q_{45}$, which is a known constant, and equation (6) becomes:

$$Q_{45} = \mathrm{XOR}(Q_{44}, Q_{43}, Q_{42}) \lll 9$$
$$Q_{45} = (Q_{45} \oplus V) \lll 9 \tag{6'}$$

This last equation is actually a system of linear equations over the bits of Q_{45}; it is easy to check whether it is satisfiable, and to compute the solutions (see Appendix B for an optimization). From Q_{45}, we compute Q_{44} by (6) and m_3 by (5), and we know that this particular choice of m_3 will give the right value for Q_{46}, and we will have $H_2 = \overline{H}_2$.

Simplifications. We have introduced two extra conditions to simplify the equations:

$$Q_1 = \mathbb{1} \tag{C_1}$$
$$Q_{41} \boxplus m_{11} \boxplus K_2 = \mathbb{0}\,. \tag{C_2}$$

$(C_1) : Q_1 = \mathbb{1}$ and $(C_2) : Q_{41} \boxplus m_{11} = \mathbb{0}$. (C_1) can only be satisfied statistically, by computing about 2^{32} initial messages and keeping the good ones, but this cost can be amortized over the many choices of Q_{32}, m_2, and m_1. On the other hand, (C_2) can be satisfied by choosing an appropriate m_1 when Q_{32} and m_2 have been chosen:

$$Q_{40} = (Q_{36} \boxplus \mathrm{XOR}(Q_{39}, Q_{38}, Q_{37}) \boxplus m_1 \boxplus K_2) \lll 3 \tag{9}$$
$$Q_{41} = (Q_{37} \boxplus \mathrm{XOR}(Q_{40}, Q_{39}, Q_{38}) \boxplus m_9 \boxplus K_2) \lll 9 \tag{10}$$

The choice of Q_{41} gives Q_{40} by (10), which gives m_1 by (9). Conversely, with this choice of m_1 (C_2) will be satisfied. Every initial message can now be used

Algorithm 2. Partial Pseudo Preimage

```
Input: $\overline{H}_0, \overline{H}_2, \overline{IV}_2$
Output: $M, IV$ st. $H_0 = \overline{H}_0$, $H_2 = \overline{H}_2$ and $IV_2 = \overline{IV}_2$
Running Time: $2^{32}$
loop                                                      ▷ Expect 1 iteration
    Choose an initial message with $Q_1 = \mathbb{1}$     ▷ $2^{96}$ possibilities
    for all $Q_{32}$ do                                   ▷ $2^{32}$ iterations
        Compute $Q_{33}, Q_{34}, Q_{35}$.
        Choose $m_2$ so that $Q_{-2} = \overline{IV}_2$.   ▷ $Q_{-2}$ is $IV_2$
        Compute $Q_{36}, Q_{37}, Q_{38}$, $Q_{39}$.
        Choose $m_1$ so that $Q_{41} = -m_{11} - K_2$.
        Compute $Q_{40}, Q_{41}, Q_{42}$, $Q_{43}$.
        Choose $m_3$ so that $Q_{46} = \overline{H}_2 \boxminus Q_{-2}$.   ▷ $Q_{46} \boxplus Q_{-2}$ is $H_2$
        Compute $Q_{44}, Q_{45}, Q_{46}, Q_{47}$, and $Q_{-1}, Q_{-2}, Q_{-3}$.
        Choose $m_0$ so that $Q_{-4} = \overline{H}_0 \boxminus Q_{44}$.   ▷ $Q_{44} \boxplus Q_{-4}$ is $H_0$
        if $m_0$ matches $Q_{32}$ then                    ▷ OK with prob. $2^{-32}$
            return
        end if
    end for
end loop
```

with 2^{64} choices of Q_{32} and m_2, so we still have some extra degree of freedom: we can use the freedom of m_2 to choose the value of Q_{-2}. In the end we can choose both Q_{-2} and Q_{46} (hence H_2) for an amortized cost of one compression function.

Partial Pseudo-Preimage. When we put this all together, we can compute m_3 so that $H_2 = \overline{H}_2$ for an amortized cost of 1 compression function, and we can also choose IV_2. The full algorithm, given in Algorithm 2, is a partial pseudo-preimage attack, which is 2^{32} times more efficient than exhaustive search. It should be repeated about 2^{64} times to find a full pseudo-preimage, and we have enough different initial messages for that. Note that Algorithm 2 finds pseudo-preimages on (H_0, H_2), but if we change a little bit the end of the algorithm we can have pseudo-preimages on (H_1, H_2) or (H_2, H_3) just as easily. See Appendix A for an example of a partial pseudo-preimage.

3 Preimage of the Full MD4

To extend this attack to the full MD4, we will use an idea similar to the unbalanced meet-in-the-middle attack of Lai and Massey [12]. We compute many pseudo-preimages of $\overline{H}$, we hash many random messages, and we use the birthday paradox to meet in the middle. If we have a pseudo-preimage attack with complexity 2^s, the generic attack uses the pseudo-preimage attack $2^{(n-s)/2}$ times starting from the target digest $\overline{H}$ (we assume there is no problem with the padding), and hashes $2^{(n+s)/2}$ random blocks, starting from the standard IV.

Thanks to the birthday paradox, we expect one match. This gives a preimage attack with complexity $2^{1+(n+s)/2}$. In our case this would be 2^{113} (we have $s = 96$), but we will show how to use some specific properties of our pseudo-preimage attack to build a preimage attack with complexity 2^{102}.

3.1 The Padding

First, we need to handle the padding in the last block. When looking for a padded message of b blocks, we will use a message length of $512b - 65$ bits. The last block is correctly padded if and only if $m_{15} = 0$, $m_{14} = 512b - 65$, and $m_{13}^{[0]} = 1$. The condition on m_{15} is easy to satisfy since we can choose m_{15} in the initial message, and on the other hand m_{14} depends only on $\mathbb{C}$:

$$\begin{aligned} Q_{27} &= (Q_{23} \boxplus \mathrm{MAJ}(Q_{26}, Q_{25}, Q_{24}) \boxplus m_{14} \boxplus k_{27}) \lll 13 \\ \mathbb{C} &= (\mathbb{C} \boxplus \mathbb{C} \boxplus m_{14} \boxplus K_1) \lll 13 \\ m_{14} &= \mathbb{C} \ggg 13 \boxminus \mathbb{C} \boxminus \mathbb{C} \boxminus K_1 \end{aligned}$$

Thus, we just run an exhaustive search over $\mathbb{C}$, and we expect to find one value that gives the correct m_{14}. Similarly, we have $m_{13} = \mathbb{C} \ggg 13 \boxminus \mathbb{C} \boxminus \mathbb{C} \boxminus K_1 = m_{14}$, so the condition $m_{13}^{[0]} = 1$ will be satisfied.

Note that we can not set a size that is a multiple of 8 this way since $m_{14} = m_{13}$ is used both as the padding and as the length. If we really need to use a message made of bytes and not of bits, we can build a second-preimage attack by reusing the last block of the original message (and keeping the same padding).

When searching for the last block, we only have 2^{32} initial messages available. We will not chose the value of IV_2, but keep m_2 as a degree of freedom. Each initial message can be used to compute 2^{64} related messages with $H_2 = \overline{H}_2$. There is probability of $1 - 1/e \approx 0.63$ that at least one of these messages will give the full correct hash, so we might have to repeat this a few times. The extra freedom will come from the message length: if we change the number of blocks b, this gives us a new m_{14} and a new $\mathbb{C}$, and we can try again to find a padding block.

We start with $b = 34$ or $b = 18$ (see next section), and increase b until we find a padding block. Note that some values of b will give no suitable value of $\mathbb{C}$, but this is not a problem. Additionally, if one wants to choose a prefix for the preimage attack, one just has to start with a bigger b.

3.2 Layered Hash Tree

To improve over the basic meet-in-the-middle attack, we will use an extra property of our pseudo-preimage attack on the compression function: we need a workload of 2^s (in our case, $s = 96$) to find a pseudo-preimage of a single target value, but if have a set of k target values (with some extra conditions on the set), we can find a pseudo-preimage of *one of them* in time $2^s/k$. This is because our pseudo-preimage attack is based on the repetition of a partial pseudo-preimage attack, where the remaining bits are random. Thus, we can find $2k$ pseudo-preimages in time 2^{s+1}, and if we can also make sure that the pseudo-preimage set satisfy the extra conditions, we can iterate this operation. We will start with a set $\mathcal{H}_0$ of size 1, and after $n-s$ iterations we have a set $\mathcal{H}_{n-s}$ of size 2^{n-s}, which we use for the unbalanced meet-in-the-middle. The resulting structure is shown in figure 3. In the end, we will find a preimage in time $2(n-s)2^s + 2^s$, using a memory of size $\mathcal{O}(2^{n-s})$.

A similar idea based on multi-target pseudo-preimage was used by Mendel and Rijmen to attack HAS-V [14]. In that attack, they could run a multi-target pseudo-preimage attack on a set of size 2^s (this is not possible in our case), and this result in an attack with time complexity 2^{s+1} and a memory requirement of $\mathcal{O}(2^s)$.

Our attack against MD4 can be used as a multi-target pseudo-preimage attack following Algorithm 3, if the target set $\mathcal{H}$ satisfies the following extra properties:

– $\{\overline{H}_2 : \overline{H} \in \mathcal{H}\}$ is a singleton: a single m_3 can be used for the whole set;
– $|\mathcal{H}| \leq 2^{64}$: the loop of line 11 is negligible.

Since our algorithm allows us to choose the value of $\overline{IV}_2$ in the pseudo-preimages, we can build the pseudo-preimage set so that the extra conditions are still satisfied.

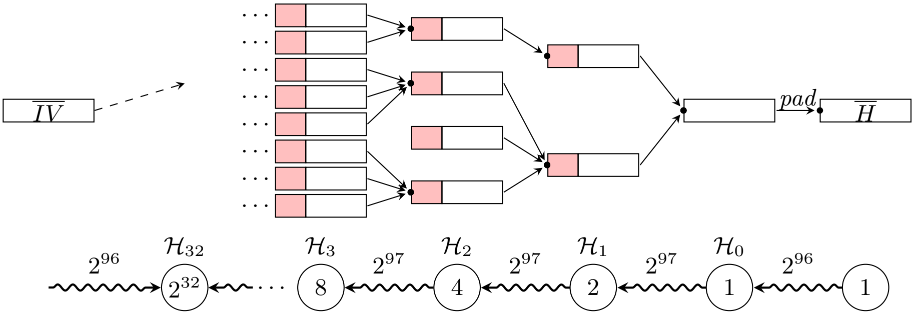


Fig. 3. Preimage attack on the full MD4 with a layered hash tree

The tree costs $32 \times 2^{97} = 2^{102}$ to build, and gives 2^{32} pseudo-preimages of $\overline{H}$; this is significantly better than the generic attack which find 2^t pseudo-preimages in time 2^{96+t}. In the forward step, we compute the hashes of 2^{96} random messages, and we expect to find a match thanks to the birthday paradox. The full preimage search has time complexity 2^{102}, and require a memory of about 2^{33} message blocks to store the tree (we do not have to store the 2^{96} hashes in the forward step).

Tweaking the Tree. We can tweak the parameters of the tree so as to slightly improve the attack. First, instead of doubling the size of the set at each iteration, we can triple it (the length of the preimage starts from 23, and the cost of the attack is about $2^{101.92}$) or quadruple it (the length of the preimage starts from 18, and the cost of the attack is about 2^{102}).

Algorithm 3. Multi-Target Pseudo Preimage

```
Input: $\overline{IV}_2$, $\overline{H}_2$ and a target set $\mathcal{H}$ st. $\forall \overline{X} \in \mathcal{H}, \overline{X}_2 = \overline{H}_2$.
Output: preimage set $\mathcal{I}$ st. $\forall (IV, M) \in \mathcal{I},\ F(IV, M) \in \mathcal{H}$ and $IV_2 = \overline{IV}_2$.
Running Time: $2^{97}$
while $|\mathcal{I}| < 2|\mathcal{H}|$ do                                  ▷ Expect $2^{65}$ iterations
    Choose an initial message with $Q_1 = \mathbb{1}$                   ▷ $2^{96}$ possibilities
    for all $Q_{32}$ do                                               ▷ $2^{32}$ iterations
        Compute $Q_{33}, Q_{34}, Q_{35}$.
        Choose $m_2$ so that $Q_{-2} = \overline{IV}_2$.                ▷ $Q_{-2}$ is $IV_2$
        Compute $Q_{36}, Q_{37}, Q_{38}, Q_{39}$.
        Choose $m_1$ so that $Q_{41} = -m_{11} - K_2$.
        Compute $Q_{40}, Q_{41}, Q_{42}, Q_{43}$.
        Choose $m_3$ so that $Q_{46} = \overline{H}_2 \boxminus Q_{-2}$.     ▷ $Q_{46} \boxplus Q_{-2}$ is $H_2$
        Compute $Q_{44}, Q_{45}, Q_{46}, Q_{47}$, and $Q_{-1}, Q_{-2}, Q_{-3}$.
        for all $\overline{H} \in \mathcal{H}$ st. $H_3 = \overline{H}_3$, $H_4 = \overline{H}_4$ do   ▷ Expect $2^{-64}|\mathcal{H}|$ values
            Choose $m_0$ so that $Q_{-4} = \overline{H}_0 \boxminus Q_{44}$.   ▷ $Q_{44} \boxplus Q_{-4}$ is $H_0$
            if $m_0$ matches $Q_{32}$ then                          ▷ OK with prob. $2^{-32}$
                Add the solution to $\mathcal{I}$
            end if
        end for
    end for
end while
```

We can also replace the layered tree by another structure. We start with a set of 1 target value, and every time we find a pseudo-preimage of one element of the set, we add it to the set. The first pseudo-preimage will cost 2^{96}, the second one $2^{96}/2$, then $2^{96}/3$ and so on... the set will have size 2^{32} after an expected workload of:

$$2^{96} \sum_{k=1}^{2^{32}} \frac{1}{k} \leq 2^{96}(\ln 2^{32} + 1) \leq 2^{100.54}.$$

In this case, we do not control the length of the preimage, so we will use an expendable message [3,9] in the forward step.

4 Conclusion

Our attack on MD4 is still theoretical due to the high complexity, but it shows that MD4 is even weaker than we thought. Our attack relies on the absorption property of some of the Boolean functions, and exploits the message expansion. It is the first preimage attack on the full MD4 and it is much less efficient than Dobbertin's attack on a two round version [6].

We did not find any direct application of the attack on the compression function, but constructions relying on the partial one-wayness of cMD4 should be carefully analysed: our attack might be practical depending on the exact assumptions made on cMD4.

This attack reduces the security margin of other members of the MD4, but it is not a direct threat. The features introduced in later members of the family make the attack unsuitable:

- The rounds function of MD5, SHA-1, and SHA-2 have a much better diffusion that MD4 due to the summation of Q_{i-1} to compute Q_i (we can not absorb a difference);
- The number of rounds is more important;
- The message expansion in the SHA family is much harder to control.

Acknowledgement

Part of this work is supported by the Commission of the European Communities through the IST program under contract IST-2002-507932 ECRYPT, and by the French government through the Saphir RNRT project.

References

1. Cramer, R.J.F. (ed.): EUROCRYPT 2005. LNCS, vol. 3494. Springer, Heidelberg (2005)
2. De, D., Kumarasubramanian, A., Venkatesan, R.: Inversion Attacks on Secure Hash Functions Using SAT Solvers. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 377–382. Springer, Heidelberg (2007)
3. Dean, R.D.: Formal Aspects of Mobile Code Security. PhD thesis, Princeton University (January 1999)
4. den Boer, B., Bosselaers, A.: An Attack on the Last Two Rounds of MD4. In: Feigenbaum, J. (ed.) CRYPTO 1991. LNCS, vol. 576, pp. 194–203. Springer, Heidelberg (1992)
5. Dobbertin, H.: Cryptanalysis of MD4. J. Cryptology 11(4), 253–271 (1998)
6. Dobbertin, H.: The First Two Rounds of MD4 are Not One-Way. In: Vaudenay, S. (ed.) FSE 1998. LNCS, vol. 1372, pp. 284–292. Springer, Heidelberg (1998)
7. Fouque, P.A., Leurent, G., Nguyen, P.: Automatic Search of Differential Path in MD4. In: ECRYPT Hash Worshop – Cryptology ePrint Archive, Report 2007/206 (2007), http://eprint.iacr.org/
8. Haller, N.: The S/KEY One-Time Password System. RFC 1760 (Informational) (February 1995)
9. Kelsey, J., Schneier, B.: Second Preimages on n-Bit Hash Functions for Much Less than 2^n Work. In: [1], pp. 474–490.
10. Klima, V.: Tunnels in Hash Functions: MD5 Collisions Within a Minute. Cryptology ePrint Archive, Report 2006/105 (2006), http://eprint.iacr.org/
11. Knudsen, L.R., Mathiassen, J.E.: Preimage and Collision Attacks on MD2. In: Gilbert, H., Handschuh, H. (eds.) FSE 2005. LNCS, vol. 3557, pp. 255–267. Springer, Heidelberg (2005)
12. Lai, X., Massey, J.L.: Hash Function Based on Block Ciphers. In: Rueppel, R.A. (ed.) EUROCRYPT 1992. LNCS, vol. 658, pp. 55–70. Springer, Heidelberg (1993)
13. Mendel, F., Rechberger, C., Rijmen, V.: Update on SHA-1. In: Menezes, A. (ed.) CRYPTO 2007. LNCS, vol. 4622. Springer, Heidelberg (2007), http://rump2007.cr.yp.to/

14. Mendel, F., Rijmen, V.: Weaknesses in the HAS-V Compression Function. In: Nam, K.-H., Rhee, G. (eds.) ICISC 2007. LNCS, vol. 4817, pp. 335–345. Springer, Heidelberg (2007)
15. Muller, F.: The MD2 Hash Function Is Not One-Way. In: Lee, P.J. (ed.) ASIACRYPT 2004. LNCS, vol. 3329, pp. 214–229. Springer, Heidelberg (2004)
16. Rivest, R.L.: The MD4 Message Digest Algorithm. In: Menezes, A., Vanstone, S.A. (eds.) CRYPTO 1990. LNCS, vol. 537, pp. 303–311. Springer, Heidelberg (1991)
17. Rogaway, P.: Formalizing Human Ignorance. In: Nguyên, P.Q. (ed.) VIETCRYPT 2006. LNCS, vol. 4341, pp. 211–228. Springer, Heidelberg (2006)
18. Rogaway, P., Shrimpton, T.: Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In: Roy, B.K., Meier, W. (eds.) FSE 2004. LNCS, vol. 3017, pp. 371–388. Springer, Heidelberg (2004)
19. Sasaki, Y., Wang, L., Ohta, K., Kunihiro, N.: New Message Difference for MD4. In: Biryukov, A. (ed.) FSE 2007. LNCS, vol. 4593, pp. 329–348. Springer, Heidelberg (2007)
20. Vaudenay, S.: On the Need for Multipermutations: Cryptanalysis of MD4 and SAFER. In: Preneel, B. (ed.) FSE 1994. LNCS, vol. 1008, pp. 286–297. Springer, Heidelberg (1995)
21. Wang, X., Lai, X., Feng, D., Chen, H., Yu, X.: Cryptanalysis of the Hash Functions MD4 and RIPEMD. In: [1], pp. 1–18
22. Wang, X., Yin, Y.L., Yu, H.: Finding Collisions in the Full SHA-1. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 17–36. Springer, Heidelberg (2005)
23. Wang, X., Yu, H.: How to Break MD5 and Other Hash Functions. In: [1], pp. 19–35
24. Yu, H., Wang, G., Zhang, G., Wang, X.: The Second-Preimage Attack on MD4. In: Desmedt, Y.G., Wang, H., Mu, Y., Li, Y. (eds.) CANS 2005. LNCS, vol. 3810, pp. 1–12. Springer, Heidelberg (2005)

A A Partial Pseudo-preimage of MD4

Here is an example of a partial pseudo-preimage of MD4. We ran our algorithm with $\overline{H}_1 = 0, \overline{H}_2 = 0$, and $\overline{IV}_2 = 0$. It would cost 2^{64} hash evaluations to find this message by exhaustive search, but our algorithm finds it in about 20 minutes on a desktop computer.

IV			
72 fa 31 aa	a0 6e 27 95	00 00 00 00	13 c9 dc ce
Message block			
8e 34 9e ad	6c 36 1e 1c	21 b7 0e bd	14 1e 98 d9
79 67 c3 19	d7 3c 6a 19	d7 3c 6a 19	d7 3c 6a 19
14 61 85 33	14 61 85 33	14 61 85 33	14 61 85 33
58 13 27 05	58 13 27 05	58 13 27 05	17 46 57 27
MD4			
34 5e 59 ae	c5 6a 3b 8e	00 00 00 00	00 00 00 00

The internal state variables for this message are given by:

Q_{-4}=0xaa31fa72	Q_{-3}=0xcedcc913	Q_{-2}=0x00000000	Q_{-1}=0x95276ea0
Q_{0} =0x1545809d	Q_{1} =0xffffffff	Q_{2} =0xa1bdf692	Q_{3} =0x1a9925ec
Q_{4} =0xa8473548	Q_{5} =0x92125811	Q_{6} =0x9b4aaa1d	Q_{7} =0x00a73054
Q_{8} =0x6ef7f38b	Q_{9} =0xa3789cab	Q_{10} =0x3de0878e	Q_{11} =0x2f9cbd24
Q_{12} =0x0ffc6391	Q_{13} =0x1e2a88f4	Q_{14} =0x1e83b396	Q_{15} =0x1e83b396
Q_{16} =0xb5062a71	Q_{17} =0x1e83b396	Q_{18} =0x1e83b396	Q_{19} =0x1e83b396
Q_{20} =0x51547062	Q_{21} =0x1e83b396	Q_{22} =0x1e83b396	Q_{23} =0x1e83b396
Q_{24} =0x3b4aa594	Q_{25} =0x1e83b396	Q_{26} =0x1e83b396	Q_{27} =0x1e83b396
Q_{28} =0x6f4786bc	Q_{29} =0x1e83b396	Q_{30} =0x1e83b396	Q_{31} =0x24db97dc
Q_{32} =0x84d9f63d	Q_{33} =0xc9a584fe	Q_{34} =0x475e7886	Q_{35} =0x508d517f
Q_{36} =0x79ca3034	Q_{37} =0x3bd701b4	Q_{38} =0x980fef11	Q_{39} =0x9784cf50
Q_{40} =0xc8f3a1b1	Q_{41} =0x5da0b34b	Q_{42} =0x5fa99919	Q_{43} =0x2d166b40
Q_{44} =0x042763c2	Q_{45} =0x312336ed	Q_{46} =0x00000000	Q_{47} =0xf913fc25

Note that MD4 uses a little-endian convention to convert a sequence of bytes to a sequence of words, and that the order of the words in the IV and in the hash is not the same as in the internal state.

B Solving the Equation $x = (x \oplus V) \lll 9$

In Section 2.2 we find that Q_{45} has to be the solution of the following equation:

$$x = (x \oplus V) \lll 9. \tag{1}$$

where V is a constant that depends on the choices made on the previous steps of the algorithm. We have to solve this equation 2^{96} times for each pseudo-preimage, so we want to solve it very efficiently (it should cost less that one evaluation of cMD4).

We can write this equation as a linear system over the bits of x and V:

$$(1) \iff \begin{cases} x^{[0]} = x^{[23]} \oplus V^{[23]} \\ x^{[1]} = x^{[24]} \oplus V^{[24]} \\ \ldots \\ x^{[31]} = x^{[22]} \oplus V^{[22]} \end{cases}$$

And we can express each bit of x as a function of $x^{[0]}$ and V:

$$\iff \begin{cases} x^{[9]} = x^{[0]} \oplus V^{[0]} \\ x^{[18]} = x^{[0]} \oplus V^{[0]} \oplus V^{[9]} \\ \ldots \\ x^{[23]} = x^{[0]} \oplus V^{[0]} \oplus V^{[9]} \oplus V^{[18]} \oplus V^{[27]} \cdots \oplus V^{[14]} \\ x^{[0]} = x^{[0]} \oplus V^{[0]} \oplus V^{[9]} \oplus V^{[18]} \oplus V^{[27]} \cdots \oplus V^{[14]} \oplus V^{[23]} \end{cases}$$

The system is consistent if and only if the last equation holds, *i.e.* $\bigoplus_{i=0}^{31} V^{[i]} = 0$. In this case we have a first solution x_0 given by

$$\begin{cases} x_0^{[\ 0]} = 0 \\ x_0^{[\ 9]} = V^{[0]} \\ x_0^{[18]} = V^{[0]} \oplus V^{[9]} \\ \dots \\ x_0^{[23]} = V^{[0]} \oplus V^{[9]} \oplus V^{[18]} \oplus V^{[27]} \dots \oplus V^{[14]} \end{cases}$$

and a second solution $x_1 = x_0 \oplus \mathbb{1}$. Note that the expression of the bits of x_0 is linear in the bits of V: $x_0 = \varphi(V)$. We will split V into 4 bytes, $V = V_3||V_2||V_1||V_0$, and we precompute 4 tables (each one contains 256 words):

$$T_0[u] = \varphi(0||0||0||u) \quad T_1[u] = \varphi(0||0||u||0) \quad T_2[u] = \varphi(0||u||0||0) \quad T_3[u] = \varphi(u||0||0||0)$$

Then we have $x_0 = \varphi(V) = T_0[V_0] \oplus T_1[V_1] \oplus T_2[V_2] \oplus T_3[V_3]$. We can solve the equation, with only:

- the computation of the parity of the hamming weight of V;
- 4 table look-ups when there is a solution.

Improved Indifferentiability Security Analysis of chopMD Hash Function

Donghoon Chang[1,⋆] and Mridul Nandi[2]

[1] Center for Information Security Technologies (CIST)
Korea University, Seoul, Korea
dhchang@cist.korea.ac.kr
[2] CINVESTAV-IPN, Mexico City
mridul.nandi@gmail.com

Abstract. The classical design principle Merkle-Damgård [13,6] is scrutinized by many ways such as Joux's multicollision attack, Kelsey-Schneier second preimage attack etc. In TCC'04, Maurer *et al.* introduced a strong security notion called as "indifferentiability" for a hash function based on a compression function. The classical design principle is also insecure against this strong security notion whereas chopMD hash is secure with the security bound roughly $\sigma^2/2^s$ where s is the number of chopped bits and σ is the total number of message blocks queried by a distinguisher. In case of $n = 2s$ where n is the output size of a compression function, the value σ to get a significant bound is $2^{s/2}$ which is the birthday complexity, where the hash output size is s-bit. In this paper, we present an improved security bound for chopMD. The improved bound shown in this paper is $(3(n-s)+1)q/2^s + q/2^{n-s-1} + \sigma^2/2^{n+1}$ where q is the total number of queries. In case of $n = 2s$, chopMD is indifferentiably-secure if $q = \mathrm{O}(2^s/(3s+1))$ and $\sigma = \mathrm{O}(2^{n/2})$ which are beyond the birthday complexity. We also present a design principle for an n-bit hash function based on a compression function $f : \{0,1\}^{2n+b} \rightarrow \{0,1\}^n$ and show that the indifferentiability security bound for this hash function is roughly $(3n+1)\sigma/2^n$. So, the new design of hash function is second-preimage and r-multicollision secure as long as the query complexity (the number of message blocks queried) of an attacker is less than $2^n/(3n+1)$ or $2^{n(r-1)/r}$ respectively.

1 Introduction

In TCC 2004, Maurer *et al.* [11] introduced the notion of indifferentiability which is more stronger notion than classical indistinguishability security notion. They have shown that if a cryptosystem $\mathcal{P}(\mathcal{G})$ based on a random oracle $\mathcal{G}$ is secure then the security of $\mathcal{P}(H^{\mathcal{F}})$ based on Merkle-Damgård (MD) [13,6] hash function

[⋆] The first author was supported by the MIC(Ministry of Information and Communication), Korea, under the ITRC(Information Technology Research Center) support program supervised by the IITA(Institute of Information Technology Advancement) (IITA-2008-(C1090-0801-0025)).

K. Nyberg (Ed.): FSE 2008, LNCS 5086, pp. 429–443, 2008.

H with a random oracle [1,15] as an underlying compression function, is secure provided the hash function is indifferentiable. Informally, $H^{\mathcal{F}}$ is indifferentiable from random oracle if there is no efficient attacker (or distinguisher) which can distinguish $\mathcal{F}$ and the hash function based on it from a random oracle R and an efficient simulator of $\mathcal{F}$. Here R is a random oracle with (finite) domain and range same as that of H. In case of Indistinguishability, the distinguisher only needs to tell apart H from $\mathcal{G}$ without any help of oracle $\mathcal{F}$. Thus, the notion of indifferentiability is stronger and it is important when we consider attacks on a cryptosystem based on some ideal primitive where the attacker has some access on the computation of the primitive. In the case of hash function $H^{\mathcal{F}}$, the attacker can also compute $\mathcal{F}$ as it is a random oracle which can be computed publicly. On the other hand, if the attacker does not have that access (to the random oracle) then merely indistinguishability will suffice to preserve the security of the cryptosystem.

In Crypto 2005, Coron *et al.* [5] proved that the classical MD iteration is not indifferentiable with random oracle when the underlying compression function is random oracle. They have also stated indifferentiability for chopMD, prefix-free MD (or pfMD), NMAC construction, HMAC construction, and provided a bound for these as $\mathrm{O}(\sigma^2/2^n)$ where σ is the total number of message blocks queried by a distinguisher, the hash output size is n, and the number of chopped bits is n. Thus, according to their claim, chopMD is secure in this strong notion as long as the total number of message blocks queried is $\sigma = \mathrm{O}(2^{n/2})$. In Asiacrypt 2006, Chang *et al.* [4] also have provided a concrete security analysis of the many indifferentiable hash constructions. They have provided a security analysis for double length hash function based on prefix free padding. In Asiacrypt 2006, Bellare and Ristenpart [2] proposed an indifferentiably-secure domain extension called by EMD which also preserves pseudorandomness and collision resistance. Very recently in Asiacrypt 2007, Hirose *et al.* [7] introduced an indifferentiably-secure domain extension called by MDP which also preserves pseudorandomness, collision resistance and unforgeability. All of these constructions have bounds of the form of birthday collision probability. Recently in Crypto 2007, Maurer and Tessaro [12] firstly presented a construction which has security beyond the birthday barrier. Table 1 summarizes the security bound of above constructions.

Our Results. In this paper, we prove a better bound of chopMD which is beyond the birthday bound. We prove that chopMD is secure if $\sigma = \mathrm{O}(2^{n/2})$ and the number of queries $Q = \mathrm{O}(\mathbf{min}(2^{n-s-1}, \frac{2^s}{3(n-s)+1}))$, where n is the output length of the compression function and s is the chopped bit length and σ is the total number of message blocks queried. When $s = n/2$ our bound shows that chopMD is secure as long as the number of queries is less that $2^s/(3s+1)$ which is better than the original proposal (where security is guaranteed only when the number of queries is less than $2^{s/2}$). As a result we propose a wide pipe version of MD-hash function which is second-preimage and r-multicollision secure as long as the query complexity (the number of message blocks queried) of an attacker is less than $2^n/(3n+1)$ or $2^{n(r-1)/r}$ respectively. This hash function is more

Table 1. Comparison of Indifferentiable Security when *the hash output size* is s and the chopped bit size is s and σ is the total number of message blocks queried by a distinguisher. Note that q is less than σ.

Domain Extensions	The value σ to get a significant bound
chopMD [5] prefix-free MD [4,5] NMAC construction [5] HMAC construction [5] EMD [2] MDP [7]	$2^{s/2}$: the Birthday Bound
prefix-free chopMD [12]	2^s : Beyond the Birthday Bound
chopMD [This paper]	$2^s/(3s+1)$: Beyond the Birthday Bound

efficient to the Lucks' [10] wide pipe hash design as our hash function does not need the post-processor.

Organization. In section 2, we first state some important definitions and results related to our paper. We state an important result known as strong interpolation theorem in this section. Then in Section 3, we provide a concrete and improved security analysis for chopMD. As an application of the improved security analysis of chopMD, we propose a secure chopDBL hash design in section 4. Finally, we conclude.

2 Some Notations and Results

Counting. Let $\mathcal{F} := \text{Func}(n+b, n)$, the set of all functions $f : \{0,1\}^n \times \{0,1\}^b \to \{0,1\}^n$. It is easy to see that $|\mathcal{F}| = 2^{n2^{n+b}}$. Now, for any distinct a_i's, the number of functions f such that $f(a_1) = z_1, \cdots, f(a_q) = z_q$ is exactly $2^{n(2^{n+b}-q)}$ because, the outputs of q elements are fixed and the rest $(2^{n+b} - q)$ many outputs can be chosen in $(2^n)^{(2^{n+b}-q)}$ many ways. Thus, $\Pr_{\mathsf{u}}[\mathsf{u}(a_1) = z_1, \cdots, \mathsf{u}(a_q) = z_q] = \frac{1}{2^{nq}}$ where u is the uniform random function on $\mathcal{F}$ (an uniform random variable taking values on $\mathcal{F}$).

Inequalities. $\mathbf{P}(m, r) = m(m-1)\cdots(m-r+1)$ where $0 \le r \le m$. By our convention, $\mathbf{P}(m, 0) = 1$. We state two inequalities which will be used in this paper.

[ineq-1] For any $0 \le a_i \le 1$, $\prod_{i=1}^{k}(1 - a_i) \ge 1 - \sum_{i=1}^{k} a_i$. One can prove it by induction on k.

[ineq-2] $\mathbf{P}(m - x, r) \ge m^r \times (1 - \frac{(x+r)^2}{2m})$ where $m \ge x + r$. This is followed from ineq-1, by choosing $a_i = \frac{x+i}{m}, 0 \le i \le r - 1$.

MD-hash. We fix an initial value IV $\in \{0,1\}^n$ throughout the paper. Given a function $f : \{0,1\}^{n+b} \to \{0,1\}^n$ we define

$$\text{MD}^f(m_1, \cdots, m_\ell) = f(f(\cdots f(f(\text{IV}, m_1), m_2), \cdots), m_\ell)$$

where $m_i \in \{0,1\}^b$. MD^f is popularly known as Merkle-Damgård hash function with underlying compression function f. We define $\mathrm{MD}^f(\lambda) = \mathrm{IV}$ where λ is the empty string. Given $p = (m_1, \cdots, m_\ell) \in (\{0,1\}^b)^\ell$ with $\ell \geq 1$ we define

- $\mathrm{last}(p) = m_\ell$.
- If $\ell \geq 2$ we write $\mathrm{cut}(p) = (m_1, \cdots, m_{\ell-1})$, otherwise $\mathrm{cut}(m_1) = \lambda$.
- Note that $p = (\mathrm{cut}(p), \mathrm{last}(p))$ and $\mathrm{MD}^f(p) = f(\mathrm{MD}^f(\mathrm{cut}(p)), \mathrm{last}(p))$.

chopMD. For $0 \leq s \leq n$ we define $\mathrm{chop}_s(x) = x_R$ where $x = x_L \parallel x_R$ and $|x_L| = s$. In this paper, we fix $0 < s < n$ and define $\mathrm{chopMD}^f(M) = \mathrm{chop}_s(\mathrm{MD}^f(M))$.

Padding. Note that both MD and chopMD have domain $(\{0,1\}^b)^+$. We write $||M|| = k$ if $M \in (\{0,1\}^b)^k$ and k is called as the number of blocks of M. We say M' is a prefix of M if $M', M \in (\{0,1\}^b)^+$ and $M = M' \parallel x$ for some $x \in (\{0,1\}^b)^*$. We say any injective function $\mathsf{pad} : \{0,1\}^* \to (\{0,1\}^b)^+$ as a padding rule. A padding rule pad is called a *prefix free* if $M_1 \neq M_2 \Rightarrow \mathsf{pad}(M_1)$ is not a prefix of $\mathsf{pad}(M_2)$. For any such prefix-free padding rule pad, pfMD is defined as follows. $\mathrm{pfMD}^f_{\mathsf{pad}}(M) = \mathrm{MD}^f(\mathsf{pad}(M))$. We also write $\mathrm{choppfMD}^f_{\mathsf{pad}}(M) = \mathrm{chop}_s(\mathrm{MD}^f(\mathsf{pad}(M)))$.

View. In this paper we consider a distinguisher $\mathcal{A}$ which has access of two oracles $\mathcal{O}_1$ and $\mathcal{O}_2$. We assume that $\mathcal{A}$ is deterministic and computationally unbounded[1]. We assume that all queries are distinct and it makes at most Q_i queries to the oracle $\mathcal{O}_i$. Suppose $\mathcal{A}$ makes M_i as $\mathcal{O}_1$-query and obtains responses h_i, $1 \leq i \leq q_1$. Similarly, the tuple of all query-responses of $\mathcal{O}_2$ is $((x_1, m_1, z_1), \cdots, (x_{q_2}, m_{q_2}, z_{q_2}))$. The combined tuple $v = ((M_1, h_1), \cdots, (M_{q_1}, h_{q_1}), (x_1, m_1, z_1), \cdots, (x_{q_2}, m_{q_2}, z_{q_2}))$ is called as the *view* of $\mathcal{A}$. We also denote $v_{\mathcal{O}_1, \mathcal{O}_2}$ to specify that the view is obtained after interacting with $\mathcal{O}_1$ and $\mathcal{O}_2$. We also denote i-th query-response pair by (X_i, Y_i), where $X_i = (x_j, m_j)$ or $X_i = M_j$ for a j. So we can define the first i query-response pairs of the tuple v by $v_i = ((X_1, Y_1), \cdots, (X_i, Y_i))$.

Advantage. Let $\mathtt{F}_i, \mathtt{G}_i$ be probabilistic oracle algorithms. We define advantage of the distinguisher $\mathcal{A}$ at distinguishing $(\mathtt{F}_1, \mathtt{F}_2)$ from $(\mathtt{G}_1, \mathtt{G}_2)$ as

$$\mathbf{Adv}_{\mathcal{A}}((\mathtt{F}_1, \mathtt{F}_2), (\mathtt{G}_1, \mathtt{G}_2)) = |\Pr[\mathcal{A}^{\mathtt{F}_1, \mathtt{F}_2} = 1] - \Pr[\mathcal{A}^{\mathtt{G}_1, \mathtt{G}_2} = 1]|.$$

Theorem 1. (Strong Interpolation Theorem) *If there is a set of good views* $\mathcal{V}_{\mathrm{good}}$ *such that*

1. *for all* $v \in \mathcal{V}_{\mathrm{good}}$, $\Pr[v_{\mathtt{F}_1, \mathtt{F}_2} = v] \geq (1 - \varepsilon) \times \Pr[v_{\mathtt{G}_1, \mathtt{G}_2} = v]$ *and*
2. $\Pr[v_{\mathtt{G}_1, \mathtt{G}_2} \in \mathcal{V}_{\mathrm{good}}] \geq 1 - \varepsilon'$

then for any $\mathcal{A}$ *we have* $\mathbf{Adv}_{\mathcal{A}}((\mathtt{F}_1, \mathtt{F}_2), (\mathtt{G}_1, \mathtt{G}_2)) \leq \varepsilon + \varepsilon'$.

[1] Computationally unbounded deterministic algorithms are as powerful as randomized algorithms.

Proof. Intuitively, a view of $\mathcal{A}^{\mathtt{G}_1,\mathtt{G}_2}$ is good with probability at least $1-\varepsilon'$. Moreover, $\mathcal{A}$ obtains a good view v with almost same probability for both pairs of oracles up to a factor of $(1-\varepsilon)$. Then intuitively the distinguishing advantage of $\mathcal{A}$ should be bounded by $\varepsilon+\varepsilon'$. More precisely, we prove it as in below where $\mathcal{V}^1$ denotes the set of all views v such that $\mathcal{A}$ returns 1 after obtaining the view[2]. $\mathcal{V}^0$ denotes the set of all views v such that $\mathcal{A}$ doesn't returns 1 after obtaining the view. And let $\alpha(v) = \Pr[\mathcal{A}(v_{i-1}) = X_i \text{ for all } 0 \le i \le q_1 + q_2]$. Our proof is directly from the idea explained in [3].

$$\begin{aligned}
&\Pr[\mathcal{A}^{\mathtt{G}_1,\mathtt{G}_2} = 1] - \Pr[\mathcal{A}^{\mathtt{F}_1,\mathtt{F}_2} = 1] \\
&\quad = \textstyle\sum_{v\in\mathcal{V}^1\cap\mathcal{V}_{\text{good}}} \alpha(v)\Pr[v_{\mathtt{G}_1,\mathtt{G}_2} = v] + \sum_{v\in\mathcal{V}^1\setminus\mathcal{V}_{\text{good}}} \alpha(v)\Pr[v_{\mathtt{G}_1,\mathtt{G}_2} = v] \\
&\qquad - \textstyle\sum_{v\in\mathcal{V}^1\cap\mathcal{V}_{\text{good}}} \alpha(v)\Pr[v_{\mathtt{F}_1,\mathtt{F}_2} = v] - \sum_{v\in\mathcal{V}^1\setminus\mathcal{V}_{\text{good}}} \alpha(v)\Pr[v_{\mathtt{F}_1,\mathtt{F}_2} = v] \\
&\quad \le \varepsilon' + \textstyle\sum_{v\in\mathcal{V}^1\cap\mathcal{V}_{\text{good}}} \alpha(v)\Pr[v_{\mathtt{G}_1,\mathtt{G}_2} = v] - \sum_{v\in\mathcal{V}^1\cap\mathcal{V}_{\text{good}}} \alpha(v)\Pr[v_{\mathtt{F}_1,\mathtt{F}_2} = v] \\
&\quad \le \varepsilon' + \textstyle\sum_{v\in\mathcal{V}^1\cap\mathcal{V}_{\text{good}}} \alpha(v)(\Pr[v_{\mathtt{G}_1,\mathtt{G}_2} = v] - \Pr[v_{\mathtt{F}_1,\mathtt{F}_2} = v]) \\
&\quad \le \varepsilon' + \varepsilon \textstyle\sum_{v\in\mathcal{V}^1\cap\mathcal{V}_{\text{good}}} \alpha(v)\Pr[v_{\mathtt{G}_1,\mathtt{G}_2} = v] \\
&\quad \le \varepsilon' + \varepsilon \textstyle\sum_{v\in\mathcal{V}^1\cap\mathcal{V}_{\text{good}}} \Pr[v_{\mathtt{G}_1,\mathtt{G}_2} = v] \\
&\quad \le \varepsilon' + \varepsilon.
\end{aligned}$$

$$\begin{aligned}
&\Pr[\mathcal{A}^{\mathtt{F}_1,\mathtt{F}_2} = 1] - \Pr[\mathcal{A}^{\mathtt{G}_1,\mathtt{G}_2} = 1] = \Pr[\mathcal{A}^{\mathtt{G}_1,\mathtt{G}_2} \neq 1] - \Pr[\mathcal{A}^{\mathtt{F}_1,\mathtt{F}_2} \neq 1] \\
&\quad = \textstyle\sum_{v\in\mathcal{V}^0\cap\mathcal{V}_{\text{good}}} \alpha(v)\Pr[v_{\mathtt{G}_1,\mathtt{G}_2} = v] + \sum_{v\in\mathcal{V}^0\setminus\mathcal{V}_{\text{good}}} \alpha(v)\Pr[v_{\mathtt{G}_1,\mathtt{G}_2} = v] \\
&\qquad - \textstyle\sum_{v\in\mathcal{V}^0\cap\mathcal{V}_{\text{good}}} \alpha(v)\Pr[v_{\mathtt{F}_1,\mathtt{F}_2} = v] - \sum_{v\in\mathcal{V}^0\setminus\mathcal{V}_{\text{good}}} \alpha(v)\Pr[v_{\mathtt{F}_1,\mathtt{F}_2} = v] \\
&\quad \le \varepsilon' + \textstyle\sum_{v\in\mathcal{V}^0\cap\mathcal{V}_{\text{good}}} \alpha(v)\Pr[v_{\mathtt{G}_1,\mathtt{G}_2} = v] - \sum_{v\in\mathcal{V}^1\cap\mathcal{V}_{\text{good}}} \alpha(v)\Pr[v_{\mathtt{F}_1,\mathtt{F}_2} = v] \\
&\quad \le \varepsilon' + \textstyle\sum_{v\in\mathcal{V}^0\cap\mathcal{V}_{\text{good}}} \alpha(v)(\Pr[v_{\mathtt{G}_1,\mathtt{G}_2} = v] - \Pr[v_{\mathtt{F}_1,\mathtt{F}_2} = v]) \\
&\quad \le \varepsilon' + \varepsilon \textstyle\sum_{v\in\mathcal{V}^0\cap\mathcal{V}_{\text{good}}} \alpha(v)\Pr[v_{\mathtt{G}_1,\mathtt{G}_2} = v] \\
&\quad \le \varepsilon' + \varepsilon \textstyle\sum_{v\in\mathcal{V}^0\cap\mathcal{V}_{\text{good}}} \Pr[v_{\mathtt{G}_1,\mathtt{G}_2} = v] \\
&\quad \le \varepsilon' + \varepsilon. \qquad \blacksquare
\end{aligned}$$

Indifferentiability

We give a brief introduction of indifferentiability and state significance of it. The following definition is a slightly modified version of the original definition [11,5], where the condition that the maximum number of message blocks queried by a distinguisher is σ is not described.

Definition 1. *[11] A Turing machine C with oracle access to an ideal primitive $\mathcal{F}$ is said to be $(t_{\mathcal{A}}, t_S, q, \sigma, \varepsilon)$-indifferentiable from an ideal primitive $\mathcal{G}$ if there exists a simulator S such that for any distinguisher $\mathcal{A}$ it holds that :*

$$\mathbf{Adv}_{\mathcal{A}}((C,\mathcal{F}),(\mathcal{G},S)) = |\Pr[\mathcal{A}^{C,\mathcal{F}} = 1] - \Pr[\mathcal{A}^{\mathcal{G},S} = 1]| < \varepsilon$$

The simulator S is an interactive algorithm which has oracle access to $\mathcal{G}$ and runs in time at most t_S. The distinguisher $\mathcal{A}$ runs in time at most $t_{\mathcal{A}}$ and makes at most q queries. The total message blocks queried by $\mathcal{A}$ is at most σ.

[2] Since $\mathcal{A}$ is deterministic algorithm the output of $\mathcal{A}$ is completely determined by the view.

The following Theorem [11] due to Maurer *et al.* is related to this paper. We explain the theorem for random oracle model of hash functions. Suppose a hash function (in some design of iteration) H based on a random oracle (or an ideal cipher) $\mathcal{F}$ is indifferentiable from a random oracle $\mathcal{G}$. Then a cryptosystem $\mathcal{P}$ based on the random oracle $\mathcal{G}$ is at least as secure as the cryptosystem $\mathcal{P}$ based on the hash function H in the random oracle model (or an ideal cipher model) $\mathcal{F}$. Here, $\mathcal{F}$ is the underlying compression function of H (or block-cipher in case of block cipher based hash function). The original theorem as stated below is a more general statement.

Theorem 2. *[11] Let $\mathcal{P}$ be a cryptosystem with oracle access to an ideal primitive $\mathcal{G}$. Let H be an algorithm such that $H^{\mathcal{F}}$ is indifferentiable from $\mathcal{G}$. Then cryptosystem $\mathcal{P}$ is at least as secure in the $\mathcal{F}$ model with algorithm H as in the $\mathcal{G}$ model.*

In this paper we consider $\mathcal{G}$ and $\mathcal{F}$ as the arbitrary input length random oracle R and the fixed input length random oracle f, respectively. And C is the chopMD hash function. If chopMD^f is $(t_{\mathcal{A}}, t_S, q, \sigma, \varepsilon)$-indifferentiable from the random oracle R, we also say that *the indifferentiability insecurity bound* of chopMD^f is ε.

3 Improved Indifferentiability Analysis of chopMD

Coron *et al.* [5] stated MD hash function is not indifferentiability-secure whereas prefix free MD construction or chopMD construction in random oracle (or in ideal cipher model) is secure against indifferentiability attack. In [5], they had proved the following statement for a distinguisher which makes queries whose total number of message blocks is σ. And u is the random oracle from the set of all $n+b$ bits to the set of n bits.

1. The indifferentiability insecurity for pfMD$^{\mathsf{u}}_{\mathsf{pad}}$ is upper bounded by $\mathrm{O}(\sigma^2/2^n)$ where pad is any prefix-free padding.
2. The indifferentiability insecurity for chopMD$^{\mathsf{u}}_{pad}$ is upper bounded by O $(\sigma^2/2^s)$.

Very recently, Maurer and Tessaro considered the combination of prefix free MD and chopMD [12], i.e., choppfMD$^{\mathsf{u}}_{\mathsf{pad}}$. They proved that the indifferentiability insecurity for this combination is bounded by $\mathrm{O}(\sigma^2/2^n)$. This is an improved bound compare to the bound for chopMD. Since choppfMD outputs $n-s$ bits, the security bound is beyond the birthday barrier. A prefix-padding may cost extra overhead in terms of efficiency and designs. In this section, we show that the the prefix-padding is not necessary to obtain the similar kind of bound. In other words, we provide an improved bound of chopMD and the improved bound stated in this paper is $(3(n-s)+1)q/2^s + q/2^{n-s-1} + \sigma^2/2^{n+1}$ where q denotes the the maximum number of queries for two oracles and σ is the total number of message blocks queried by a distinguisher. If we choose $s = n/2$ then to have a significant advantage, the total number of blocks of all queries should be at least $2^s/(3s+1)$ which is far beyond the birthday attack complexity.

The organization of section 3 is as follows. In subsection 3.1, we define a set of good views $\mathcal{V}^r_{\text{good}}$ and give a lower bound of $\Pr[v_{\mathtt{F}_1,\mathtt{F}_2} = v]$ for all $v \in \mathcal{V}^r_{\text{good}}$, where $\mathtt{F}_1$ is chopMD$^{\mathtt{u}}$ and $\mathtt{F}_2$ is $\mathtt{u}$. In subsection 3.2, we give an upper bound of $\Pr[v_{\mathtt{G}_1,\mathtt{G}_2} = v]$ for all $v \in \mathcal{V}^r_{\text{good}}$, where $\mathtt{G}_1$ is the random oracle R and $\mathtt{G}_2$ is the simulator S^R described in subsection 3.2. In subsection 3.3, we compute ε and ε' such that for all $v \in \mathcal{V}_{\text{good}}$, $\Pr[v_{\mathtt{F}_1,\mathtt{F}_2} = v] \geq (1-\varepsilon) \times \Pr[v_{\mathtt{G}_1,\mathtt{G}_2} = v]$ and $\Pr[v_{\mathtt{G}_1,\mathtt{G}_2} \in \mathcal{V}_{\text{good}}] \geq 1 - \varepsilon'$. Finally, based on Theorem 1 (strong interpolation theorem), we conclude in Theorem 3 that the indifferentiability insecurity bound of chopMD$^{\mathtt{u}}$ is $\varepsilon^* = \varepsilon + \varepsilon'$.

3.1 Interpolation Probability of chopMD and Its Underlying Random Oracle

We first provide a lower bound on the number of functions when some inputs-outputs of f and MD^f are known. More precisely, we want to compute the number of functions f such that

$$\text{MD}^f(M_j) = h_j \text{ and } f(a_i) = z_i, 1 \leq j \leq q_1, 1 \leq i \leq q_2$$

where $a_i \in \{0,1\}^{n+b}$ are distinct, $M_j \in (\{0,1\}^b)^+$ are distinct. Intuitively, we say the above set of relations is irreducible (see definition 2 in below) if $\text{MD}^{\mathcal{O}_2}(M_i)$ is not determined from $\mathcal{O}_2(x_1,m_1) = z_1, \cdots, \mathcal{O}_2(x_{q_2},m_{q_2}) = z_{q_2}$ and $\text{MD}^{\mathcal{O}_2}(M_j) = h_j$ for all $j \neq i$. Thus, q_1 many outputs of MD^f add q_1 more restrictions on the outputs of f besides q_2 many input-output relations of f. Hence the number of functions f should be close to $2^{n(2^{n+b}-q_1-q_2)}$. In lemma 1 we will show that the number of functions is at least $(1-\nu) \times 2^{n(2^{n+b}-q_1-q_2)}$ for some positive ν (stated in the lemma 1) close to zero. The above statement is also equivalent to

$$\Pr_{\mathtt{u}}[\text{MD}^{\mathtt{u}}(M_j) = h_j, \mathtt{u}(a_i) = z_i \ \forall 1 \leq j \leq q_1, 1 \leq i \leq q_2] \geq \frac{1}{2^{n(q_1+q_2)}} \times (1-\nu).$$

Definition 2. *The set of relations*

$$\begin{aligned}&\text{MD}^{\mathcal{O}_2}(M_1) = h_1, \cdots, \text{MD}^{\mathcal{O}_2}(M_{q_1}) = h_{q_1},\\ &\mathcal{O}_2(x_1,m_1) = z_1, \cdots, \mathcal{O}_2(x_{q_2},m_{q_2}) = z_{q_2} \quad \cdots\cdots \text{ (rel-A)}\end{aligned}$$

is said to be irreducible *if* $M_1, \cdots, M_{q_1} \in (\{0,1\}^b)^+$ *are distinct,* (x_1,m_1), $\cdots$, $(x_{q_2},m_{q_2}) \in \{0,1\}^{n+b}$ *are distinct,* $h_1, \cdots, h_{q_1} \in \{0,1\}^n$ *are distinct from* x_i*'s and* IV *and finally the value of* $\text{MD}^{\mathcal{O}_2}(M_i)$ *is not determined from the relations* $\mathcal{O}_2(x_1,m_1) = z_1$, $\cdots$, $\mathcal{O}_2(x_{q_2},m_{q_2}) = z_{q_2}$. *A tuple of elements* $v = ((M_1,h_1), \cdots, (M_{q_1},h_{q_1}), (x_1,m_1,z_1), \cdots, (x_{q_2},m_{q_2},z_{q_2}))$ *is* irreducible *if the above* rel-A *is irreducible*[3].

[3] From the definition it is clear that irreducibility of the relation does not depend on the choice of the functions or oracles $\mathcal{O}_1$ and $\mathcal{O}_2$. This only depends on M_j's, h_j's, (x_i,m_i)'s and z_i's, $1 \leq j \leq q_1$ and $1 \leq i \leq q_2$.

Remark 1. Intuitively, it says that there is no redundant relation in rel-A. All the inputs of $\mathcal{O}_1$ and $\mathcal{O}_2$ are distinct. $\mathcal{O}_1(M_i) = \mathrm{MD}^{\mathcal{O}_2}(M_i)$ is also not determined from the relations $\mathcal{O}_2(x_1, m_1) = z_1, \cdots, \mathcal{O}_2(x_{q_2}, m_{q_2}) = z_{q_2}$. Moreover, as h_i's are distinct from x_i's and IV, $\mathrm{MD}^{\mathcal{O}_2}(M_i)$ is also not determined from $\mathcal{O}_2(x_1, m_1) = z_1, \cdots, \mathcal{O}_2(x_{q_2}, m_{q_2}) = z_{q_2}$ and $\mathrm{MD}^{\mathcal{O}_2}(M_j) = h_j$ for all $j \neq i$.

Lemma 1. *Let a tuple* $v = ((M_1, h_1), \cdots, (M_{q_1}, h_{q_1}), (x_1, m_1, z_1), \cdots, (x_{q_2}, m_{q_2}, z_{q_2}))$ *be irreducible then the number of functions* f *such that*

1. $\mathrm{MD}^f(M_1) = h_1, \cdots, \mathrm{MD}^f(M_{q_1}) = h_{q_1}$ *and*
2. $f(x_1, m_1) = z_1, \cdots, f(x_{q_2}, m_{q_2}) = z_{q_2}$.

is at least $\frac{|\mathcal{F}|}{2^{n(q_1+q_2)}} \times (1 - \frac{\sigma^2}{2^{n+1}})$ *where* σ *is the total number of message blocks queried. In other words,*

$$\Pr_{\mathtt{u}}[\mathrm{MD}^{\mathtt{u}}(M_1)=h_1, \cdots, \mathrm{MD}^{\mathtt{u}}(M_{q_1})=h_{q_1}, \mathtt{u}(x_1, m_1) = z_1, \cdots, \mathtt{u}(x_{q_2}, m_{q_2}) = z_{q_2}]$$

$$\geq \frac{1}{2^{n(q_1+q_2)}} \times (1 - \frac{\sigma^2}{2^{n+1}}).$$

Proof. See the appendix. ∎

Now we compute the joint probability for $\mathrm{chopMD}^{\mathtt{u}}$ and $\mathtt{u}$. The next lemma is analogue version of Lemma 1 for chopMD hash function instead of MD hash function. Here, we allow collisions among outputs of chopMD. Intuitively, if $\mathrm{chopMD}^{\mathtt{u}}$ behaves as an uniform random function then $\Pr_{\mathtt{u}}[\mathrm{chopMD}^{\mathtt{u}}(M_j) = y_j, \mathtt{u}(x_i, m_i) = z_i, 1 \leq j \leq q_1, 1 \leq i \leq q_2]$ ideally should be $\frac{1}{2^{nq_2+(n-s)q_1}}$. Since $\mathrm{chopMD}^{\mathtt{u}}(M_j) = y_j$ has some influence on the intermediate computations we would rather expect a probability close to the above probability. In lemma 2 we show that the for a given choices of inputs and outputs satisfying some conditions (stated in the lemma 2) the above probability is at least $\frac{1-\Delta}{2^{nq_2+(n-s)q_1}}$ for some positive Δ (defined in the lemma 2) which is close to zero for reasonable choices of parameters.

Lemma 2. *The number of functions* f *such that*

1. $\mathrm{chopMD}^f(M^1_1) = \cdots = \mathrm{chopMD}^f(M^1_{r_1}) = y^1, \cdots, \mathrm{chopMD}^f(M^t_1) = \cdots = \mathrm{chopMD}^f(M^t_{r_t}) = y^t$ *and*
2. $f(x_1, m_1) = z_1, \cdots, f(x_{q_2}, m_{q_2}) = z_{q_2}$.

is at least $|\mathcal{F}| \times \frac{1-\Delta}{2^{nq_2+(n-s)q_1}}$ *where*

$$\Delta = \frac{r(q_1+q_2)}{2^s} + \frac{\sigma^2}{2^{n+1}}, \quad r = \mathrm{max}_i r_i, \quad \sum_i r_i = q_1.$$

Here, σ *is the total number of message blocks queried.* M^i_j*'s are distinct elements from* $(\{0,1\}^b)^+$ *such that the value of* $\mathrm{MD}^f(M_i)$ *is not determined from the relations* $f(x_1, m_1) = z_1, \cdots, f(x_{q_2}, m_{q_2}) = z_{q_2}$*. The values of* (x_i, m_i)*'s are distinct elements from* $\{0,1\}^n \times \{0,1\}^b$*. In terms of probability, we have*

$$\Pr_{\mathtt{u}}[\mathrm{chopMD}^{\mathtt{u}}(M_j^i) = y^i, \mathtt{u}(x_1, m_1) = z_1, \cdots, \mathtt{u}(x_{q_2}, m_{q_2}) = z_{q_2},\ \forall i, j] \geq \frac{1-\mathit{\Delta}}{2^{nq_2+(n-s)q_1}}.$$

Proof. See the appendix. ■

Definition 3. *A view $v = ((M_1, h_1), \cdots, (M_{q_1}, h_{q_1}),(x_1, m_1, z_1), \cdots, (x_{q_2}, m_{q_2}, z_{q_2}))$ is said to be r-good if (x_i, m_i)'s are distinct, M_j's are distinct,* $\mathrm{MD}^{\mathcal{O}_2}(M_j)$ *is not determined from the relations $\mathcal{O}_2(x_i, m_i) = z_i$ and there is no r-multicollision in* $\mathrm{chop}(z_i)$*'s and h_i's. The set of all r-good views is denoted by $\mathcal{V}^r_{\mathrm{good}}$.*

By using lemma 2 we have similar result for $\mathrm{chopMD}^{\mathtt{u}}$ and $\mathtt{u}$.

Proposition 1. *For any r-good view $v = ((M_1, h_1), \cdots, (M_{q_1}, h_{q_1}),(x_1, m_1, z_1), \cdots, (x_{q_2}, m_{q_2}, z_{q_2}))$, the probability that v is a view when $\mathcal{A}$ is interacting with* $\mathrm{chopMD}^{\mathtt{u}}$ *and* $\mathtt{u}$*, is at least $\frac{1-\mathit{\Delta}}{2^{nq_2+(n-s)q_1}}$ where $\mathit{\Delta} = \frac{r(q_1+q_2)}{2^s} + \frac{\sigma^2}{2^{n+1}}$ and σ is the total number of message blocks queried.*

3.2 Interpolation Probability of a Simulator and Random Oracle

Now we define a simulator S which almost behaves as a random oracle. Moreover, for an $(n-s)$-bit outputting random oracle R, responses of MD^S will match with R. By the notation $x \in_R A$ we mean that x is chosen uniformly from A and it is independent with all previously defined random variables.

Definition of Simulator

Initialization :

1. A partial function $e_1 : \{0,1\}^{n+b} \rightarrow \{0,1\}^n$ initialized as empty,
2. a partial function $e_1^* = \mathrm{MD}^{e_1} : (\{0,1\}^b)^* \rightarrow \{0,1\}^n$ initialized as $e_1^*(\lambda) = \mathrm{IV}$.
3. a set $C = \{\mathrm{IV}\}$.

On query $S^R(x, m)$:

```
001 if ($e_1(x, m) = x'$)
        return $x'$;

002 else if ($\exists\ M', e_1^*(M') = x$)
        $y = R(M', m)$;
        choose $w \in_R \{0,1\}^s \setminus \{w' : w' \parallel y \in C \cup \{x\}\}$;
        define $e_1(x, m) = z := w \parallel y$;
        define $C = C \cup \{x, z\}$;
        define $e_1^*(M', m) = z$;
        return $z$;
```

```
003 else
        $y \in_R \{0,1\}^{n-s}$;
        choose $w \in_R \{0,1\}^s \setminus \{w' : w' \parallel y \in C \cup \{x\}\}$;
        define $e_1(x,m) = z := w \parallel y$;
        define $C = C \cup \{x,z\}$;
        return $z$;
```

In 002, we have $w \in_R \{0,1\}^s \setminus \{w' : w' \parallel y \in C \cup \{x\}\}$. This is not possible if and only if the above set becomes empty. Note that after i^{th} query the size of C is less than or equal to $(2i+1)$. Thus we assume that q_2, the maximum number of queries to the simulator (and hence for oracle $\mathcal{O}_2$) satisfies the condition $2q_2 + 2 < 2^s$ equivalently $q_2 \leq 2^{s-1} - 2$.

Some Important Observations

Distinct Query. Suppose $\mathcal{A}^{\mathcal{O}_1,\mathcal{O}_2}$ is an oracle algorithm where $\mathcal{O}_1 = R$ and $\mathcal{O}_2 = S^R$. Note that S^R responses identically in identical queries and so does R. Same property is true for chopMD^f and f. Hence we assume that all queries to $\mathcal{O}_1$ and $\mathcal{O}_2$ are distinct.

chopMD$^{\mathbf{S}}$= R. All responses of S are distinct and distinct from IV and the first n-bits of all previous S-queries. Whenever MD$^S(M)$ is computable from the all previous query-responses, we have chopMD$^S(M)$= $R(M)$. Thus, chopMD$^{\mathcal{O}_2}(M)$ = $\mathcal{O}_1(M)$ whenever chopMD$^{\mathcal{O}_2}(M)$ is computable from $\mathcal{O}_2$ query-responses only. Obviously this is true when $\mathcal{O}_1$ = chopMD^f and $\mathcal{O}_2 = f$. Thus, we assume that $\mathcal{A}$ do not make any $\mathcal{O}_1$-query which is computable from the previous query-responses of $\mathcal{O}_2$. More particularly, we can remove all those $\mathcal{O}_1$-queries from the final view which are computable from the query-responses of $\mathcal{O}_2$.

Distribution. Because of the above two assumptions, the last $(n-s)$ bits of outputs of $S^R(\cdot)$ and outputs of $R(\cdot)$ are uniformly and independently distributed over the set $\{0,1\}^{n-s}$. By our assumption, whenever line002 is executed, $\mathcal{A}$ does not make (M',m)-query to R. Thus, *the output distribution of* $R(\cdot)$ *and* $S(\cdot)$ *are independent.*

Now, a typical view of $\mathcal{A}^{\mathcal{O}_1,\mathcal{O}_2}$ is a tuple

$$v = ((M_1,h_1),\cdots,(M_{q_1},h_{q_1}),(x_1,m_1,z_1),\cdots,(x_{q_2},m_{q_2},z_{q_2}))$$

where $\mathcal{O}_1(M_j) = h_j$ and $\mathcal{O}_2(x_i,m_i) = z_i$. Moreover, (x_i,m_i)'s are distinct, M_j's are distinct and MD$^{\mathcal{O}_2}(M_j)$ is not determined from the relations $\mathcal{O}_2(x_i,m_i) = z_i$. Now we compute the joint interpolation probabilities for S and R. More precisely, $p := \Pr[R(M_j) = h_j\ \forall j$ and $S(x_i,m_i) = z_i\ \forall i]$. Since outputs of S and outputs of R are independently distributed, it is sufficient to compute the joint probabilities of S and R separately. Obviously, $\Pr[R(M_j) = h_j\ \forall j] = \frac{1}{2^{(n-s)q_1}}$. Now on i^{th} query of S, the response of (x_i,m_i) is z_i with probability at most $\frac{1}{2^{n-s}} \times \frac{1}{2^s - \ell_i}$ where

$$\ell_i = |\{k : 1 \leq k \leq q_2, \text{chop}(x_k) = \text{chop}(z_i)\}| \\ + |\{k : 1 \leq k \leq q_2, \text{chop}(z_k) = \text{chop}(z_i)\}| + 1.$$

ℓ_i is the upper bound of the size of the set $\{w' : w' \parallel y \in C \cup \{x\}\}$ appeared in the i^{th} query of S. Multiplying all these probabilities we obtain $\Pr[S(x_i, m_i) = z_i \ \forall i] \leq \frac{1}{2^{nq_2}} \times \frac{1}{1-\sum_i \ell_i/2^s}$.

It is easy to see that for any r-good view $\sum_i \ell_i \leq (2r+1)q_2$. Thus, we have proved the following result.

Proposition 2. *For any r-good view $v= ((M_1, h_1), \cdots, (M_{q_1}, h_{q_1}), (x_1, m_1, z_1), \cdots, (x_{q_2}, m_{q_2}, z_{q_2}))$, the probability that v is a view when $\mathcal{A}$ is interacting with the simulator S and a random oracle R, is at most $\frac{1}{2^{nq_2+(n-s)q_1}} \times \frac{1}{1-(2r+1)q_2/2^s}$.*

3.3 Indifferentiability Security Bound of chopMD

Now we compute ε and ε' such that for all $v \in \mathcal{V}_{\text{good}}$, $\Pr[v_{\mathtt{F}_1,\mathtt{F}_2} = v] \geq (1-\varepsilon) \times \Pr[v_{\mathtt{G}_1,\mathtt{G}_2} = v]$ and $\Pr[v_{\mathtt{G}_1,\mathtt{G}_2} \in \mathcal{V}_{\text{good}}] \geq 1-\varepsilon'$, where $\mathtt{F}_1$ is chopMD$^{\mathtt{u}}$, $\mathtt{F}_2$ is $\mathtt{u}$, $\mathtt{G}_1$ is the random oracle R and $\mathtt{G}_2$ is the the simulator S^R.

The Value of ε. By proposition 1 and 2, for all $v \in \mathcal{V}_{\text{good}}$, we have a lower bound of $\Pr[v_{\mathtt{F}_1,\mathtt{F}_2} = v]$ and an upper bound of $\Pr[v_{\mathtt{G}_1,\mathtt{G}_2} = v]$. So, we can choose $\varepsilon = \frac{(3r+1)q_2+rq_1}{2^s} + \frac{\sigma^2}{2^{n+1}}$. When $r = n-s$, $\varepsilon = \frac{(3(n-s)+1)q_2+(n-s)q_1}{2^s} + \frac{\sigma^2}{2^{n+1}}$.

The Value of ε'. Now we compute ε' such that $\Pr[v_{\mathtt{G}_1,\mathtt{G}_2} \in \mathcal{V}_{\text{good}}] \geq 1-\varepsilon'$, where $\mathtt{G}_1$ is the random oracle R and $\mathtt{G}_2$ is the simulator S^R. $v_{\mathtt{G}_1,\mathtt{G}_2} \in \mathcal{V}_{\text{good}}$ means that the view $v_{\mathtt{G}_1,\mathtt{G}_2}$ is r-good. Therefore, we have to prove that the upper bound of the probability that there is a r-multicollision among q uniformly and independently chosen $(n-s)$-bits is ε'. Let's compute this ε' as follows. Let us denote the $\mu(n-s, r, q)$ for the probability that there is a r-multicollision among q uniformly and independently chosen $(n-s)$-bits. Now it is easy to see that $\mu(n-s, r, q) \leq \frac{\binom{q}{r}}{2^{(n-s)(r-1)}}$. Now we choose $r = n-s$ and hence $\mu(n-s, r, q) \leq (q/2^{n-s-1})^r \leq q/2^{n-s-1}$ if $q \leq 2^{n-s-1}$. Since chop$(S(\cdot))$ and $R(\cdot)$ uniformly and independently distributed over $\{0,1\}^{n-s}$, a $(n-s)$-good view is obtained by $\mathcal{A}^{S,R}$ with probability at least $1 - q/2^{n-s-1}$, where $q = q_1 + q_2$. Therefore, we can choose $\varepsilon' = q/2^{n-s-1}$ when $r = n-s$.

Now, by using proposition 1 and 2 and strong interpolation theorem we obtain our following main theorem of the section. Here, $\varepsilon^* = \varepsilon + \varepsilon'$.

Theorem 3. *The chopMD construction is $(t_{\mathcal{A}}, t_S, q, \sigma, \varepsilon^*)$-indifferentiable from a random oracle, in the random oracle model for the compression function, for any $t_{\mathcal{A}}$, with $t_S = \ell \cdot \mathrm{O}(q^2)$ and $\varepsilon^* = \frac{(3(n-s)+1)q_2+(n-s)q_1}{2^s} + \frac{q}{2^{n-s-1}} + \frac{\sigma^2}{2^{n+1}} = \mathrm{O}(\frac{nq}{2^s} + \frac{q}{2^{n-s}} + \frac{\sigma^2}{2^n})$, where $q = q_1 + q_2$.*

4 chopDBL Hash Functions and Its Security Analysis

A r-multicollision for a hash function H is a r-set $\{X_1, \cdots, X_r\}$ such that $H(X_1) = \cdots = H(X_r)$. In [8] it is shown that the r-multicollision can be found in the classical MD hash function in roughly $2^{n/2}$ complexity. For a random oracle it needs [14] roughly $2^{n(r-1)/r}$ complexity. Moreover, Kelsey-Schneier [9] found a second preimage attack which needs roughly $2^{n/2}$ queries for classical MD hash function. But for a random oracle to have a second preimage attack we need at least 2^n queries. Thus MD hash function is not good in terms of multicollision and second-preimage attack. Lucks designed a wide pipe hash which is secure against these attacks.

We first define Lucks wide pipe design. In his design let $F : \{0,1\}^{w+b} \rightarrow \{0,1\}^w$ and $g : \{0,1\}^w \rightarrow \{0,1\}^n$ be two independently distributed random oracles. The wipe pipe hash [10] is defined as $g(\mathrm{MD}^F(M))$ for any padded message M. In [10], it was shown that

- the second preimage attack for the wide pipe hash needs $\min\{2^{w/2}, 2^n\}$ complexity.
- the k-multicollision for the wide pipe hash needs $\min\{2^{w/2}, 2^n\}$ complexity.

Here we show that the random oracle assumption of g is redundant. More precisely, we obtain almost similar bound when g is a simply chop function. Thus we define a chopDBL hash function as

$$\mathrm{chopDBL}^F(m_1, \cdots, m_\ell) = \mathrm{chop}_n(\mathrm{MD}^F(m_1, \cdots, m_\ell)).$$

One can compute $F : \{0,1\}^{2n+b} \rightarrow \{0,1\}^{2n}$ based on two independent random oracles $f_1, f_2 : \{0,1\}^{2n+b} \rightarrow \{0,1\}^n$ as $F(X) = f_1(X) \parallel f_2(X)$. As shown in the last section, we have an improved security analysis for chopMD. By using Theorem 3 we know that $\mathrm{chop}_n\mathrm{MD}^F$ is $2^n/(3n+1)$-indifferentiable secure.

Theorem 4. *The chopDBL construction is* $(t_{\mathcal{A}}, t_S, q, \sigma, \varepsilon)$*-indifferentiable from a random oracle, in the random oracle model for the compression function, for any* $t_{\mathcal{A}}$*, with* $t_S = \ell \cdot O(q^2)$ *and* $\varepsilon = \mathrm{O}(\frac{nq}{2^n} + \frac{q}{2^n} + \frac{\sigma^2}{2^{2n}})$.

The above theorem says that to have an indifferentiability attack we need at least $2^n/(3n+1)$ query complexity (the number of message blocks queried). Thus, if we can have second preimage attack of chopDBL with q query complexity then $q \geq 2^n/(3n+1)$. Otherwise we can distinguish chopDBL from a random oracle with less queries than $2^n/(3n+1)$. A similar argument shows that r-multicollision attack needs at least minimum of $2^{n(r-1)/r}$ and $2^n/(3n+1)$ queries. Thus in the random oracle model our new design of hash function is almost optimally secure (with respect to second preimage and multicollision).

5 Conclusion

In this paper, we present an improved security analysis for chopMD. This improved security analysis helps us how to get security beyond the birthday barrier. More precisely, we design an n-bit wide pipe hash function which has security level close to 2^n and hence we have beyond birthday barrier. The new design is much simpler and efficient. It would be interesting to see whether it preserves other properties more particularly, second preimage security.

Acknowledgement

Thank Anonymous referees for giving us valuable comments.

References

1. Bellare, M., Rogaway, P.: Random Oracles Are Practical: A Paradigm for Designing Efficient Protocols. In: 1st Conference on Computing and Communications Security, pp. 62–73. ACM Press, New York (1993)
2. Bellare, M., Ristenpart, T.: Multi-Property-Preserving Hash Domain Extension and the EMD Transform. In: Lai, X., Chen, K. (eds.) ASIACRYPT 2006. LNCS, vol. 4284, pp. 299–314. Springer, Heidelberg (2006)
3. Bernstein, D.J.: A short proof of the unpredictability of cipher block chaining (2005), http://cr.yp.to/antiforgery/easycbc-20050109.pdf
4. Chang, D., Lee, S., Nandi, M., Yung, M.: Indifferentiable Security Analysis of Popular Hash Functions with Prefix-Free Padding. In: Lai, X., Chen, K. (eds.) ASIACRYPT 2006. LNCS, vol. 4284, pp. 283–298. Springer, Heidelberg (2006)
5. Coron, J.S., Dodis, Y., Malinaud, C., Puniya, P.: Merkle-Damgard Revisited: How to Construct a Hash Function. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 430–448. Springer, Heidelberg (2005)
6. Damgard, I.B.: A design principle for hash functions. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 416–427. Springer, Heidelberg (1990)
7. Hirose, S., Park, J.H., Yun, A.: A Simple Variant of the Merkle-Damgård Scheme with a Permutation. In: Kurosawa, K. (ed.) ASIACRYPT 2007. LNCS, vol. 4833, pp. 113–129. Springer, Heidelberg (2007)
8. Joux, A.: Multicollisions in iterated hash functions. In: Franklin, M. (ed.) CRYPTO 2004. LNCS, vol. 3152, pp. 306–316. Springer, Heidelberg (2004)
9. Kelsey, J., Schneier, B.: Second pre images on n-bit hash functions for much less than 2^n work. In: Cramer, R.J.F. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 474–490. Springer, Heidelberg (2005)
10. Lucks, S.: A Failure-Friendly Design Principle for Hash Functions. In: Roy, B. (ed.) ASIACRYPT 2005. LNCS, vol. 3788, pp. 474–494. Springer, Heidelberg (2005)
11. Maurer, U., Renner, R., Holenstein, C.: Indifferentiability, Impossibility Results on Reductions, and Applications to the Random Oracle Methodology. In: Naor, M. (ed.) TCC 2004. LNCS, vol. 2951, pp. 21–39. Springer, Heidelberg (2004)
12. Maurer, U., Tessaro, S.: Domain Extension of Public Random Functions: Beyond the Birthday Barrier. In: Menezes, A. (ed.) CRYPTO 2007. LNCS, vol. 4622, pp. 187–204. Springer, Heidelberg (2007)
13. Merkle, R.C.: One way hash functions and DES. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 428–446. Springer, Heidelberg (1990)
14. Nandi, M., Stinson, D.R.: Multicollision Attacks on Some Generalized Sequential Hash Functions. Information Theory 53(2), 759–767 (2007)
15. Shannon, C.: Communication theory of secrecy systems. Bell Systems Technical Journal 28(4), 656–715 (1949)

Appendix

Proof of Lemma 1. Let D be the set of all elements from $(\{0,1\}^b)^+$ whose MD^f values are determined from the relations $f(x_1, m_1) = z_1, \cdots, f(x_{q_2}, m_{q_2})$

$= z_{q_2}$. Since v is irreducible, $M_i \notin D$ for all $1 \leq i \leq q_1$. Let P denotes the set of all nonempty prefixes of M_i's. More precisely,

$$P = \{M \in (\{0,1\}^b)^+ : M \text{ is a prefix of } M_i \text{ for some } 1 \leq i \leq q_1\}.$$

We enumerate the set $P \setminus (D \cup \{M_1, \cdots, M_{q_1}\}) := \{N_1, \cdots, N_{\sigma'}\}$. Note that, $|P| \leq \sum_i ||M_i||$. Now, we have

$$\sigma = q_2 + \sum_i ||M_i|| \ \geq \ q_2 + |P| \ \geq \ q_2 + \sigma' + q_1 := \sigma'',$$

where σ is the total number of message blocks queried. Now we choose σ' distinct elements $z'_1, \cdots, z'_{\sigma'} \in \{0,1\}^n$ which are distinct from x_i's and IV. These values will be assigned as intermediate outputs of f during the computation of $\mathrm{MD}^f(M_i)$'s. We can choose such z'_i's in at least $\mathbf{P}(2^n - q_2 - 1, \sigma')$ ways. Now given any such choices of z'_i's we count the number of functions f such that

1. $f(x_1, m_1) = z_1, \cdots, f(x_{q_2}, m_{q_2}) = z_{q_2}$,
2. $\mathrm{MD}^f(M_1) = h_1, \cdots, \mathrm{MD}^f(M_{q_1}) = h_{q_1}$ and
3. $\mathrm{MD}^f(N_1) = z'_1, \cdots, \mathrm{MD}^f(N_{\sigma'}) = z'_{\sigma'}$.

Claim: relation 1,2,3 $\Leftrightarrow$ relation 1 and $f(a_1) = h_1, \cdots f(a_{q_1}) = h_{q_1}, f(a'_1) = z'_1, \cdots, f(a'_{\sigma'}) = z'_{\sigma'}$, where (x_i, m_i)'s, a_i's and a'_i's are all distinct. Moreover, the values of a_i's and a'_is are completely determined from the tuples v=((x_1, m_1, z_1), $\cdots$, $(x_{q_2}, m_{q_2}, z_{q_2})$, (M_1, h_1), $\cdots$, (M_{q_1}, h_{q_1})) and $(z'_1, \cdots, z'_{\sigma'})$. More precisely, $a_i = (c_i, \mathrm{last}(M_i))$ where

$$\begin{aligned} c_i &= z'_j \text{ if } \mathrm{cut}(M_i) = N_j \\ &= \mathrm{IV} \text{ if } \mathrm{cut}(M_i) = \lambda \\ &= h_j \text{ if } \mathrm{cut}(M_i) = M_j \\ &= z_j \text{ if } \mathrm{MD}^f(\mathrm{cut}(M_i)) = z_j \text{ is determined from the relation 1} \end{aligned}$$

Similarly, $a'_i = (c'_i, \mathrm{last}(N_i))$ where

$$\begin{aligned} c'_i &= z'_j \text{ if } \mathrm{cut}(N_i) = N_j \\ &= \mathrm{IV} \text{ if } \mathrm{cut}(N_i) = \lambda \\ &= h_j \text{ if } \mathrm{cut}(N_i) = M_j \\ &= z_j \text{ if } \mathrm{MD}^f(\mathrm{cut}(N_i)) = z_j \text{ is determined from the relation 1} \end{aligned}$$

From the above discussion it is clear that the relations 1,2 and 3 equivalently correspond to the σ many distinct input-outputs of f. Thus the number of functions f satisfying 1,2 and 3 is exactly $2^{n(2^{n+b} - \sigma'')}$ where $\sigma'' = q_1 + q_2 + \sigma'$. By multiplying the number of choices of z'_is with $2^{n(2^{n+b} - \sigma'')}$, we obtain the number of functions satisfying 1 and 2 is at least

$$2^{n(2^{n+b} - \sigma'')} \times \mathbf{P}(2^n - q_2 - 1, \sigma') \geq \frac{|\mathcal{F}|}{2^{n(q_1+q_2)}} \times (1 - \frac{(\sigma' + q_2 + 1)^2}{2^{n+1}}) \geq \frac{|\mathcal{F}|}{2^{n(q_1+q_2)}} \times (1 - \frac{\sigma^2}{2^{n+1}}).$$

This follows from ineq-2 (stated in the beginning of the section). This proves the first part. The second part is trivial from the first part since u has uniform distribution on $\mathcal{F}$ and hence we need to divide the above quantity by $|\mathcal{F}|$. ■

Proof of Lemma 2. We denote ℓ_i as the number of pairs (x_k, m_k) such that $\mathrm{chop}(x_k) = y^i$. More precisely, $\ell_i = |\{k : 1 \le k \le q_2, \mathrm{chop}(x_k) = y^i\}|$. Since y^i's are distinct, $\ell_1 + \cdots + \ell_t \le q_2$. Now we choose $w^i_j \in \{0,1\}^s$, $1 \le j \le r_i$, $1 \le i \le t$ such that

$$h^i_j = (w^i_j \parallel y^i)\text{'s are distinct and also distinct from } x_i\text{'s and IV.} \qquad \text{(A)}$$

The number ways we can choose w^i_j's satisfying the above condition (A) is at least

$$I_1 := (2^s - \ell_1 - 1)(2^s - \ell_1 - 2)\cdots(2^s - \ell_1 - r_1)\cdots(2^s - \ell_t - 1)\cdots(2^s - \ell_t - r_t).$$

We can choose w^1_1 in $2^s - \ell_1 - 1$ ways as there are ℓ_1 many x_k's with $\mathrm{chop}(x_k) = y^1$ and chop(IV) can be equal to y^1. After choosing w^1_1 we can choose w^1_2 in $(2^s - \ell_1 - 2)$ ways and so on. Now, after choosing all $w^1_1, \cdots, w^1_{\ell_1}$ we can choose w^2_1 in $2^s - \ell_2 - 1$ ways since $y^2 \ne y^1$ and so on. Thus we have I_1 many w^i_j's with the condition (A). A straight forward simplification shows that $I_1 \ge 2^{sq_1}(1 - r(q_1 + q_2)/2^s)$ (we use the relations $\sum_i \ell_i \le q_2$, $r_i \le r$ and $\sum_i r_i = q_1$). Now for any fixed such choice of w^i_j's, the values h^i_j's are distinct from x_i's and IV. Thus, the tuple

$$v = ((x_1, m_1, z_1), \cdots, (x_{q_2}, m_{q_2}, z_{q_2}), (M^1_1, h^1_1), \cdots, (M^1_{r_1}, h^1_{r_1}), \cdots,$$
$$(M^t_1, h^t_1), \cdots, (M^t_{r_t}, h^t_{r_t}))$$

is irreducible. Hence the number of functions $f \in \mathrm{Func}(n + b, n)$ such that

1. $f(x_1, m_1) = z_1, \cdots, f(x_{q_2}, m_{q_2}) = z_{q_2}$ and
2. $\mathrm{MD}^f(M^1_1) = h^1_1, \cdots, \mathrm{MD}^f(M^1_{r_1}) = h^1_{r_1}, \cdots, \mathrm{MD}^f(M^t_1) = h^t_1, \cdots, \mathrm{MD}^f(M^t_{r_t}) = h^t_{r_t}$

is at least $\frac{|\mathcal{F}|}{2^{n(q_1+q_2)}} \times (1 - \frac{\sigma^2}{2^{n+1}})$ (by using Lemma 1). So, the number of functions satisfying the relation in this lemma is at least

$$\frac{|\mathcal{F}|}{2^{n(q_1+q_2)}} \times (1 - \frac{\sigma^2}{2^{n+1}}) \times 2^{sq_1}(1 - \frac{r(q_1+q_2)}{2^s}) \ge |\mathcal{F}| \times \frac{1 - \Delta}{2^{nq_2 + (n-s)q_1}},$$

where $\Delta = \frac{r(q_1+q_2)}{2^s} + \frac{\sigma^2}{2^{n+1}}$. The second part is followed from the first part. ■

New Techniques for Cryptanalysis of Hash Functions and Improved Attacks on Snefru[*]

Eli Biham

Computer Science Department
Technion – Israel Institute of Technology
Haifa 32000, Israel
biham@cs.technion.ac.il
http://www.cs.technion.ac.il/∼biham/

Abstract. In 1989–1990, two new hash functions were presented, Snefru and MD4. Snefru was soon broken by the newly introduced differential cryptanalysis, while MD4 remained unbroken for several more years. As a result, newer functions based on MD4, e.g., MD5 and SHA-1, became the de-facto and international standards. Following recent techniques of differential cryptanalysis for hash function, today we know that MD4 is even weaker than Snefru. In this paper we apply recent differential cryptanalysis techniques to Snefru, and devise new techniques that improve the attacks on Snefru further, including using generic attacks with differential cryptanalysis, and using virtual messages with second preimage attacks for finding preimages. Our results reduce the memory requirements of prior attacks to a negligible memory, and present a preimage of 2-pass Snefru. Finally, some observations on the padding schemes of Snefru and MD4 are discussed.

1 Introduction

Snefru [7] and MD4 [13] are two hash functions designed in 1989–1990. Soon after, an attack on Snefru based on the newly introduced differential cryptanalysis was published [2,1]. As a result, MD4 became the de-facto standard. Later, MD5 [14] and SHA-1 [8], which are improved functions of the MD4 family, replaced MD4 as the official and de-facto standards. Following the recent attacks of Wang [15] we know that collisions of MD4 can be found by hand, with complexity between 2^2 and 2^6, so that MD4 is even weaker than Snefru. Snefru has several variants, varying in the number of passes and the hash sizes. The supported hash sizes are 128 and 256 bits. The number of passes in the original 2-pass variant of Snefru is two passes, while a more secure 4-pass version is also available. After the prior attacks were published, an 8-pass version was introduced as well. This 8-pass version is still considered secure.

[*] This work was supported in part by the Israel MOD Research and Technology Unit.

K. Nyberg (Ed.): FSE 2008, LNCS 5086, pp. 444–461, 2008.

The most basic and standard attacks on hash functions are the birthday attacks. Their standard implementation requires a large memory (whose complexity is typically the same as the time complexity). Memoryless generic attacks, in which only a negligible amount of memory is required, can also be performed with the same time complexity. The most well known of those is the Floyd [4, Page 7] two-pointer cycle detection algorithm, for which there is a variant that finds a collision. Floyd algorithm is also used in other cryptographic contexts, e.g., the Pollard Rho factoring algorithm [11]. The algorithms of [12,10] are more efficient and parallelized versions. Another interesting algorithm (which is between three and five times faster than Floyd) was designed by Nivasch [9]. This algorithm uses a single pointer and a small stack, and saves the overhead of advancing two pointers, and applying the iterated function three times at each step.

Soon after the introduction of Snefru, Biham and Shamir applied their newly introduced techniques of differential cryptanalysis [2,1] to Snefru, and presented efficient second preimage and collision attacks on 2-pass Snefru, as well as many instances of collisions and second preimages. Their attack could also find second preimages and collisions of 3-pass and 4-pass Snefru, but the higher time complexities and memory requirements prevented them from implementing these attacks.

In this paper we present two main techniques that extend these attacks on Snefru: one that combines differential cryptanalysis with generic collision search techniques, in order to get the best of both worlds, i.e., the complexity of differential cryptanalysis, with the memory requirements of the generic memoryless collisions search techniques. Using this technique, the first collision of 3-pass Snefru was found. The other contribution presents a method that allows to use a second preimage attack for finding preimages of the compression function of Snefru, using specially crafted *virtual messages*, which are not preimages of the function, but on which the second preimage attack can be applied, and find a preimage of the function. Many such preimages of the compression function of 2-pass Snefru were found using this technique. The same attack can also be used for 3-pass and 4-pass Snefru, with higher complexities. We also discuss the importance of the particular padding scheme of the hash function, and how it affects the ability to find preimages of the full hash function. Finally, a very long preimage of 2-pass Snefru is presented.

The paper is organized as follows. Section 2 describes Snefru. Prior attacks on Snefru are described in Section 3. Section 4 describes generic memoryless collision search algorithms. In Section 5 memoryless collision attacks based on the combination of differential cryptanalysis with generic attacks are described, and Section 6 describes the preimage attacks on Snefru. Finally, Section 7 summarizes the paper.

2 Description of Snefru

Snefru [7] is an iterative hash function that follows the Merkle-Damgård construction [5,6,3]. It was designed to be a cryptographic hash function which hashes messages of arbitrary length into 128-bit values (a 256-bit variant based

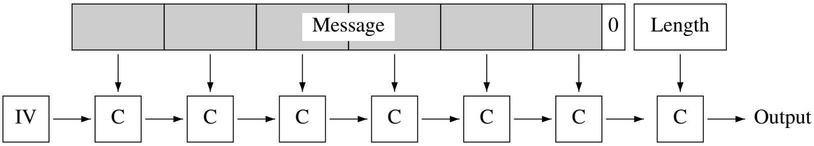


Fig. 1. The Mode of Operation of Snefru

on the same design was also introduced). Snefru uses a padding scheme that always adds an additional padding block with the length of the message (unlike the more compact padding scheme of MD4 [13], which adds another block only if necessary). Messages are divided into 384-bit blocks, $M_1, \ldots, M_{n-1}$, where the last block is padded by '0's. An additional block M_n containing only the message length (in bits) is appended. Each block is then mixed with the chaining value (initially using $IV = 0$) by a compression function C. The compression function C takes a 512-bit input composed of the chaining value and the current block, and calculates a new chaining value. More formally, $h_i = \mathrm{C}(h_{i-1} \| M_i)$ for any $1 \leq i \leq n$, where '$\|$' is the concatenation operator of bit vectors, M_i is block number i, and $h_0 = 0$. The final hash is h_n. This mode of operation is outlined in Figure 1.

The compression function is based on an invertible 512-bit to 512-bit permutation (which can be viewed as a keyless block cipher). The permutation mixes the data in two passes in the standard two-pass version of Snefru, and in four passes in the more secure four-pass Snefru. Each pass is composed of 64 mixing rounds. In round i, the least significant byte of word $i \bmod 16$ is used as an input to an 8x32-bit S box, whose 32-bit output is XORed into the two neighboring data words (all word indices are taken mod 16). After every set of 16 rounds (to which we call a *quarter*), a rotation of each of the words is performed, in order to ensure that each of the 64 bytes is used as an input to an S box once every pass (64 rounds). The output of the compression function is the XOR of the input chaining value with the last words of the output of the permutation. The details of the compression function are described in Figure 2. A complete description on Snefru, including the S boxes, can be found in [7].

3 Prior Attacks

Prior attacks on Snefru are discussed in [2,1]. These attacks include collision attacks as well as second preimage attacks. Some of the prior attacks even work if the attacker does not know the details of the S boxes. In this section we describe the main ideas and basic techniques of the attacks, concentrating on the variants with 128-bit hash values.

The second preimage attack is as follows: choose a random block-sized message and prepend the given 128-bit input chaining value to form a 512-bit input to the permutation of the compression function. We create a second message from the

```
function C (int32 input[16]) returns int32 output[4]
{
  int32 block[16];
  int32 SBoxEntry;
  int i, index, byteInWord;

  int shiftTable[4] = {16, 8, 16, 24};

  block = input;
  for index = 0 to NO_OF_PASSES-1 do {     (pass index)
    for byteInWord = 0 to 3 do {           (quarter index*4+byteInWord)
      for i = 0 to 15 do {                 (round index*64+byteInWord*16+i)
        SBoxEntry = SBOX[2*index+((i/2) mod 2)][block[i] mod 256];
        block[(i + 1) mod 16] ⊕= SBoxEntry;
        block[(i - 1) mod 16] ⊕= SBoxEntry;
      }
      for i = 0 to 15 do
        block[i] = block[i] ⋙ shiftTable[byteInWord];
    }
  }

  for i = 0 to 3 do
    output[i] = input[i] ⊕ block[15-i];

  return(output);
}
```

Fig. 2. The Compression Function of Snefru

first one by modifying two or more bytes in words 5–11, which are used as inputs to the S boxes at rounds 53–59 (i.e., those bytes that do not affect the computation before the fourth quarter). We hash both messages by the compression function and compare the outputs of the two executions. A fraction of 2^{-40} of these pairs of messages are hashed to the same value. Therefore, by hashing about 2^{41} messages we can find a second preimage. As described later in this section, the number can be greatly reduced by using more structured messages.

The characteristic used in this attack is outlined in Figure 3. In the figure each column represents a word of data and each row represents a quarter (16 rounds), where each round is marked by a thin line along the edges. The input appears at the top of the figure, and the calculation is performed downwards. The gray area in the middle represents arbitrary non-zero differences, while the white areas represent zero differences. The two thick black lines at the top-left and the bottom-right corners point to the words which are used in the calculation of the hash value by the compression function. Since both of them occur in the white part of the block, such two messages hash to the same value. In this characteristic, no difference is propagated till round 53 (fourth quarter),

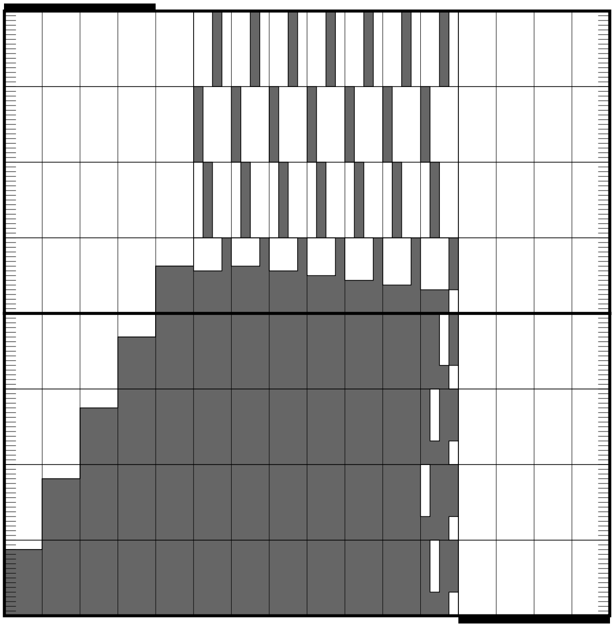


Fig. 3. Graphic Description of the Characteristic

in which the difference affects the input of the S box for the first time. The output of the S box then affects the two neighboring words, and so repeatedly in the next rounds till round 58. With some luck, which comes with probability 2^{-8}, the output of the S box of round 58 cancels the difference of the least significant byte in the next word, leading to a zero difference in the input to the S box in round 59, and then to zero differences in the inputs to the S boxes of the following rounds till the next quarter. The difference propagates again in round 68–74 (fifth quarter), and with some luck, the difference in the S box of round 75 is zero again. Similarly, if we are lucky three more times (in rounds 91, 107, and 123), then the difference of the last four words of the permutation is zero, which after XORed with the input chaining value, is leading to a collision of the output. We call this pillar of lucks a *wall* (as all lucks stand on top of each other), and the slow propagation of the differences at the left hand side a *stairs shape*. The total probability of all the five "lucks" in the wall is $2^{-8 \cdot 5} = 2^{-40}$. As the first two "lucks" can be assured deterministically by a simple selection of the changes, the actual probability is 2^{-24}. Thus, a second preimage attack using this characteristic and all these techniques has complexity $2 \cdot 2^{24} = 2^{25}$.

These ideas lead to very efficient collision attacks, by using structures of messages. We randomly choose about $2^{12.5}$ messages in which all the inputs in the white area are the same in all messages, while the messages differ in (up to seven) gray bytes. For each message, we apply the needed (deterministic) modifications to fix the difference of the "lucky" S boxes of quarters 4 and 5. We then hash all these

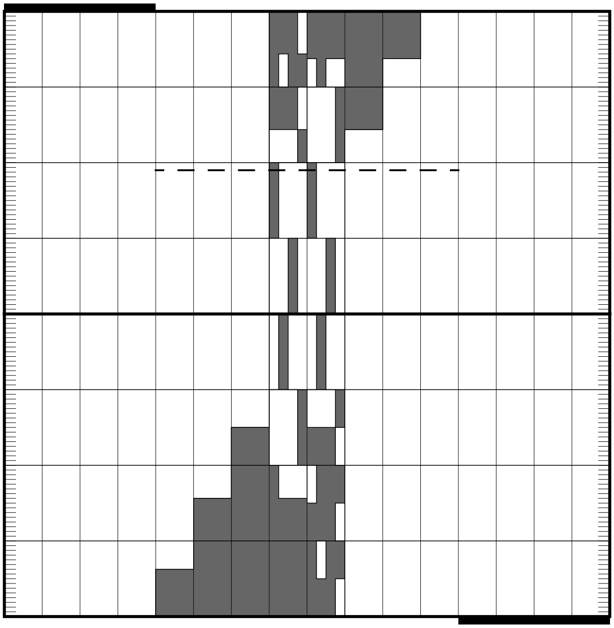


Fig. 4. A Characteristic with Modification at an Intermediate Round

messages. We get about $\frac{(2^{12.5})^2}{2} = 2^{24}$ pairs of messages which are then subjected to the second preimage attack. With high probability such a structure contains a right pair, i.e., a pair whose two messages hash to the same value. Such a pair can be easily found by sorting the $2^{12.5}$ hashed values. The memory complexity of this attack is the same as the time complexity, due to the need to keep all the computed values, for performing the tests for collisions. This attack can also be used when Snefru is considered as a black box, which hides the choice of the S boxes.

An important observation is that whenever the S boxes are known to the attacker, the modification of the bytes may be performed at an intermediate round rather than in the message itself. In this case we choose a message and partially hash it, in order to get the value of the data block at some intermediate round. Figure 4 describes such a characteristic, which modifies the data at the intermediate round denoted by the dashed line. The attack modifies the gray bytes at the marked location, rather than in the message itself. Then, the input of the permutation is calculated by performing the inverse of the compression function backwards from the marked location, and its output is calculated forward. From the input and the output of the permutation, the output of the compression function is then calculated. The remaining details of the attacks are the same as in the prior case.

Characteristics may also have differences in all the bytes of words in the marked location (rather than just one in each word). Such characteristics are very useful for longer versions of the hash function, i.e., 3-pass and 4-pass Snefru.

Table 1. Summary of the Complexities of the Prior Attacks

No. of passes	Second Preimage (time)	Collision (time & memory)
2	2^{24}	$2^{12.5}$
3	2^{56}	$2^{28.5}$
4	2^{88}–2^{112}	$2^{44.5}$–$2^{56.5}$

A summary of the prior second preimage and collision attacks on Snefru is given in Table 1.

4 Generic Memoryless Collision-Search Algorithms

In this section we briefly describe two generic memoryless collision-search algorithms.

4.1 Floyd Algorithm

Floyd algorithm [4, Page 7] traverses the graph generated by iterative application of a function f. It is used by Pollard's Rho factoring algorithm [11], and variants of which are used in the attacks on hash functions of Oorschot and Wiener [10]. In this algorithm, two pointers to the graph are used, one advances by one edge at a time, while the other advances by two at a time. At some moment, the slower pointer enters a cycle, and some time afterwards, their distance would be a multiple of the cycle size, i.e., they will both point to the same location inside the cycle. Once we identify this fact, we deduce the size of the cycle (actually a multiple of the cycle size), and an approximate information on the length of the path from the starting point to the cycle. The collision search variant of Floyd algorithm repeats the process, one pointer starts from the starting point, and the other from the reached location on the cycle, and both advance at the same speed of one edge at a time. After some time they will both point to the first location of the cycle. The previous values of these two pointers form the collision. A detailed description is given in Figure 5. The expected complexity of this attack is about $2^{m/2}$, and it calls the function f up to five times for each value in the path to the collision.

4.2 Nivasch Algorithm

In 2004, Nivasch described another cycle detection algorithm [9] that uses only a single pointer into the graph, but keeps a small stack that consists of increasing values of the vertices. At each step, the stack contains all the values in the path that satisfy the property that no smaller value exists in the path anywhere between them and the current point. Therefore, the current value is at the top of the stack, just below it resides the last value in the path that is smaller than

```
1. Let $f$ be the hash function.
2. Select some starting point $v_0$ at random.
3. $u = f(v_0)$, $v = f(f(v_0))$.
4. while $u \neq v$ do
   (a) $u = f(u)$.
   (b) $v = f(f(v))$.
5. If $v = v_0$, the starting point $v_0$ is in the cycle – stop, and try again with another
   starting point.
6. $u = v_0$.
7. repeat
   (a) $u' = u$, $v' = v$.
   (b) $u = f(u)$.
   (c) $v = f(v)$.
   until $u = v$.
8. Now $u' \neq v'$, and $f(u') = f(v')$.
```

Fig. 5. Floyd Algorithm with Collision Search

```
1. Let $f$ be the hash function.
2. Initialize a stack.
3. Select some starting point $v_0$ at random, and assign $u = v_0$.
4. while stack is empty or $u \neq$ top(stack)
   (a) Push $u$ to the stack.
   (b) Compute $u = f(u)$.
   (c) While stack is not empty and top(stack) $> u$, remove the top entry from the
       stack.
5. Once this line is reached, $u$ is the minimal value in the cycle.
```

Fig. 6. Nivasch Cycle Detection Algorithm

it, and so on. The idea is that once we reach the minimal point in the cycle the second time, the minimal point is the only point in the cycle that remains on the stack, so it is easy to identify this point. This cycle detection is described in Figure 6. The expected size of the stack is logarithmic with the number of computations of $f(u)$, which in practical cases is only a few tens up to an hundred. This algorithm is up to three times faster than Floyd's. The difference is especially meaningful when the tail of the path is larger than the cycle. A multi-stack variant of Nivasch algorithm can detect the cycle even earlier by utilizing several stacks without loss of efficiency [9]. An adaptation of the multi-stack variant of Nivasch's algorithm, with collision search (using a second pointer), which makes an optimal use of the values in the stacks, is given in Figure 7. At first reading, it is advisable to follow the cycle detection part of this algorithm up to Step 8 in order to understand the stacks method, and then follow it again with a single stack (by assuming that $S = 1$ and $g(\cdot) \equiv 0$), and ignore Steps 11–12 (which are optimization steps, and the algorithm would work correctly without them), so that only the collision search is added on top of the original algorithm.

```
1. Let $f$ be the hash function.
2. Initialize an array of $S$ stacks.
3. Let $g : \text{range}(f) \to \{0, \ldots, S-1\}$ be an efficient balanced function, e.g., a function that truncates the input to $\log_2 S$ bits.
4. Select some starting point $v_0$ at random, and assign $u = v_0$.
5. Compute $s = g(u)$.
6. Initialize a counter $c = 0$.
7. while stack[$s$] is empty or $u \neq$ top(stack[$s$])
   (a) Push the pair $(u, c)$ to stack[$s$].
   (b) Compute $u = f(u)$.
   (c) Compute $s = g(u)$.
   (d) Increment $c$.
   (e) While stack[$s$] is not empty and top(stack[$s$]).$u > u$, remove the top entry from stack[$s$].
8. Compute the size of the cycle $p = c -$top(stack[$s$]).$c$.
9. Assign $c = c_m =$ top(stack[$s$]).$c$.
10. Assign $v = v_0$ and $d = 0$.
11. For each stack[$s$], $s \in \{0, \ldots, S-1\}$
    – If there is a pair $(u', c')$ in stack[$s$] (including in popped entries that were not yet overwritten) such that $d < c' < c_m$, assign $v = u'$ and $d = c'$ (if there are several such pairs, use the one with the largest $c'$).
12. For each stack[$s$], $s \in \{0, \ldots, S-1\}$
    – If there is a pair $(u', c')$ in stack[$s$] (including in popped entries that were not yet overwritten) such that $c < c' \leq d + p$, assign $u = u'$ and $c = c'$ (if there are several such pairs, use the one with the largest $c'$).
    – Otherwise, there is a pair with a minimal $c'$ such that $c_m \leq c'$, if also $c < c'$ assign $u = u'$ and $c = c'$
13. if $c - p > d$, iteratively compute $v = f(v)$, $c - p - d$ times.
14. if $c - p < d$, iteratively compute $u = f(u)$, $d - c + p$ times.
15. If $u = v$, the starting point $v_0$ is in the cycle – stop, and try again with another starting point.
16. repeat
    (a) $u' = u$, $v' = v$.
    (b) $u = f(u)$.
    (c) $v = f(v)$.
    until $u = v$.
17. Now $u' \neq v'$, and $f(u') = f(v')$.
```

Fig. 7. Nivasch Multi-Stack Algorithm with Collision Search

5 Using Generic Algorithms for Attacking Snefru

The main drawback of the prior attacks on Snefru is the requirement for a large memory, which in the case of a 3-pass Snefru is about eight gigabytes, and in the case of a 4-pass Snefru is many thousands of terabytes.

In this section we describe a new technique which uses a generic collision search algorithm in conjunction with differential cryptanalysis. This combination

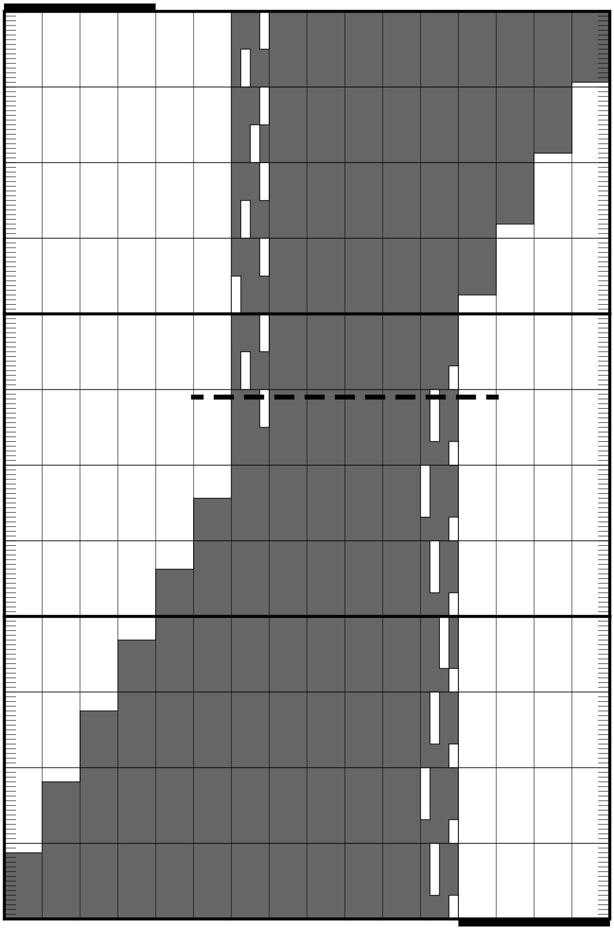


Fig. 8. A Three-Pass Characteristic

was not known in the past, and shows that seemingly unrelated techniques may be combined to form better attacks.

The collision attack on 3-pass Snefru uses the characteristic given in Figure 8. In this characteristic, more than 64 bits may be modified in the marked location. As the conditions of the first six quarters (on the marked location and above it) can be ensured with probability 1 by various simple changes to the message block, the probability of this characteristic is 2^{-56}, i.e., a second preimage can be found after about 2^{56} trials, and a collision may be found after $2^{28.5}$ trials but using $2^{28.5}$ records of memory. Let k be the number of trials required for the collision attack (e.g., $k = 2^{28.5}$ in the case of 3-pass Snefru).

We observe that the prior attack on Snefru makes various tweaks to a message block (or intermediate data) with the hope that some of the tweaks will have the same hash result. The attack can be summarized as follows:

1. Select a block, and compute the values in the location marked by a dashed line.
2. Do about k times:
   (a) Select a "random" tweak for the active bytes in marked location.
   (b) **Assign the tweak to the active bytes in the marked location.**
   (c) **Modify the required bytes to control the first six quarters.**
   (d) **Compute backwards to get the message block.**
   (e) **Compute forward to get the output of the permutation.**
   (f) **Compute the XOR of the last output words with the chaining value, resulting with the output of the compression function for that block.**
   (g) Insert to a hash table.
   (h) If a collision is found: report the collision and stop.

Though it is not impossible these days to apply this attack on 3-pass Snefru using about 8GB of RAM, this is certainly a limit on the practicality of the attack. For 4-pass Snefru the required memory size ensures that the attack would be impossible to apply for many years to follow. We would thus prefer to need a smaller amount of memory.

We observe that Steps 2b–2f can be viewed as a function f of the tweak. With this notation, the attack becomes:

1. Select a block, and compute the values in the location marked by a dashed line.
2. Do about k times:
   (a) Select a "random" tweak for the active bytes in marked location.
   (b–f) **Compute $y = f($tweak$)$.**
   (g) Insert to a hash table.
   (h) If a collision is found: report the collision and stop.

However, once we model the attack as a repetitive application of the function f and a search for a collision of the outputs, we are able to use memoryless collision search techniques instead, using the same function f.

Note that for using the memoryless algorithms, f should have the same number of input and output bits, which is reached by truncating the output to the size of the tweak. In the case of the 3-pass example, f processes 64-bit values into 64-bit values (by truncating the 128-bit output to 64 bits). We expect a random collision (of the truncated values) after 2^{32} iterations — such a collision is a false alarm. We expect a real collision, due to the differential characteristic, after about $2^{28.5}$ iterations — such a collision is a collision on the full 128-bit original output. As the probability that a random collision would occur within the first $2^{28.5}$ iterations is small, with a very high probability the collision found by the memoryless collision search algorithm is a collision of the attacked hash function.

An example collision of 3-pass Snefru found by this technique is given in Table 2. Using Nivasch algorithm, this collision was found in about half an hour on a personal computer (Intel Core Duo, 1.6GHz), using an unoptimized code.

Table 2. An Example Collision of 3-Pass Snefru (in hexadecimal)

First message:	00000000	00000000	00000000	014A2500	D5717D14	06A9DE9B
	12DB2554	304D2ECE	421F027B	063C73AD	1AF7BDC1	A1654FED
Second message:	00000000	00000000	00000000	9F713600	69B6241A	25DE987C
	D142F521	F1A56064	D9BF9D7E	E03501DA	680D062F	D136E7EA
Common compressed value:	70DE98A5	4FA2634A	E57E0F2D	7F93FCD9		

In the case of a collision of 4-pass Snefru, the time complexity remains as in the prior attack ($2^{44.5}$–$2^{56.5}$), but without the need for a huge memory.

6 Virtual Messages and Preimages of the Compression Function

6.1 A Preimage Attack on the Compression Function

In this section we describe a technique that uses the second preimage attack in order to find preimages of the compression function. The second preimage attack on 3-pass Snefru has complexity 2^{56} using the characteristic of Figure 8. In order to use the second preimage attack, we need to find a message block with the same chaining value and output of a compression function. But this message block would also form a preimage, which would cause the rest of the technique to be redundant. Moreover, it would require to find a preimage in order to find a preimage, which makes such an attack impossible.

We observe that the second preimage attack can actually find blocks with different chaining values and/or different outputs than the given message, as long as they satisfy several *compatibility* criteria related to the values of the bytes in the walls, the input chaining value, and the output of the permutation. We call such a compatible message, which is not a preimage, but can be used with the second preimage attack to find a preimage, a *virtual message*. Thus, in order to find a preimage, we only need to apply the second preimage attack on the virtual message, and control the output so that the attack will find a collision.

We emphasize that the virtual message does not have the required chaining value nor output. In some cases, it could be viewed as a combination of two parts, one corresponding to the chaining value and the wall that protects it (the top left white area of the characteristic), and one to the output and the walls that protects it (the bottom right white area of the characteristic). During the second preimage attack the two parts would be fully combined into a real preimage.

Consider the characteristic of Figure 8. During a compression of a second preimage, the original message and the second preimage should have the same values in all the white areas. In case the original message is a virtual message, the situation is more complicated, as the virtual message does not fulfill the requirements of a real original message. Instead, it should satisfy the following extra conditions

1. It should be possible to replace the input chaining value of the virtual message by the required input chaining value (the one needed by a real preimage) without affecting the other requirements (with minimal additional changes).
2. It should be possible to replace the output area after the last round by the required output (the one needed by a real preimage) without affecting the other requirements.

If those two requirements would be independent, the attack may have been easy. However, these two replacements affect the first words and last words of the data block (words 0 and 15), which in turn affect each other (cyclically). Therefore, they should be performed without disturbing each other, meaning without affecting the input to the S boxes of the first and last words (in rounds 0, 16, ..., 176 for the first word, and rounds 15, 31, ..., 191 for the last word). Therefore, it also should satisfy the following condition

3. When replacing any of the above values, no changes are allowed in the inputs of the S boxes of rounds 0, 16, ..., 176, and 15, 31, ..., 191.

We now observe that once these inputs (and corresponding outputs) of the S boxes of the last word (rounds 15, 31, ..., 191) are fixed, the input chaining value fully controls the inputs to the S boxes of rounds 0, 16, 32, and 48. In the case of round 0, the input to the S box is just one of the bytes of the chaining value. In the case of round 16 the input is an XOR of the first word of the chaining value, the output of the S box at round 1 (whose input is the XOR of the output of the S box of round 0 with another byte of the chaining value), and the fixed output of the S box of round 15. The other two cases (in rounds 32 and 48) are more complex functions of the fixed values and the input chaining value.

The search for a virtual message block starts by selecting the output of the last round (where the last four words are the value needed for the preimage), and computing backwards to receive the input chaining value and block. The probability to receive the required input chaining value is negligible (2^{-128}).

Recall that the original second preimage attack fixes all the white areas in Figure 8, and fixes the wall of the least significant bytes of word 6 in the first six quarters and the wall of the bytes that become least significant bytes of word 11 in the last eight quarters. The values that are fixed in the walls are selected to be their values in the original message.

In our case, we fix the wall of the last rounds to be equal to the message we start with (as the output of the last round has the required value), and we fix the wall of the first rounds to be compatible with both the required input chaining value and the required values of word 0 in all the quarters (which also behaves like a wall of 12 quarters height). This compatibility is not automatic, as the input chaining value reached by computing backwards is not expected to be compatible.

We can view our attack as having two sets of requirements, one ensures the expected behavior in the white left hand side (without difference compared to the message we start with), and ensures compatibility to the required initial value at that side. Similarly, the other ensures the expected behavior in the right hand side, and ensures compatibility of the output. Both are protected from each other by the three requirements mentioned above, including the walls. The virtual message block tries to emulate both requirements simultaneously, the first as if it had the required input chaining value, and the latter as if it had the required output value (while still satisfying the requirements on words 0 and 15).

This latter compatibility is achieved by replacing the chaining value resulting from the backward computation mentioned above, along with fixation of the wall of word 0, and selection of the wall of word 6 as becomes necessary. These changes in the wall in word 6 can then easily be compensated later during the second preimage attack.

Only in one of every 2^{32} trials of a backward computation, the replacement of the chaining value results with compatibility with the required chaining value, without affecting the wall of word 0. The search for the virtual message starts by about 2^{32} such trials, till a message in which we can replace the chaining value to the one we want, and in which the first four quarters of the wall at word 0 remain valid, is found. Once such a message is found, the remainder of the quarters of the wall at word 0 can be directly controlled by modifying words 4 and 5 and by changing the fixation of the wall at word 6. This direct control is very efficient. For example, assume that all the wall of word 0 but the last quarter is already selected as required, then the input to the S box in round 176 is 1-1 related to the least significant byte in the sixth quarter of the wall at word 6 (as the S boxes are byte-wise invertible).

Once all these changes are made, and the values of the wall in word 6 are decided, the computation can be performed in the forward direction from round 0 to the marked round, starting with the modified chaining value and modified values of words 4 and 5. The walls at words 0 and 6 ensure that words outside this range do not affect the values of words 0–5 at the marked location. These six words are now injected to replace the original values of these words at the marked location as received from the original backward computation.

As a result, words 0–5 are computed by this last forward computation, and fit to the change in the chaining value (the left white area). Words 12–15 remain as decided by the original backward computation. If only the wall at word 11 and the wall at word 15 are kept as decided, the result at the output is necessarily as required, independently of any changes in other words. Words 6–11 may be freely selected by the attacker, just as in the second preimage attack, and whose role is to allow the attacker to control the walls of words 6 and 11 as required.

Note that the received intermediate block, located at the marked location is now a combination of three parts. If we would take this block and compute backwards and forward to receive an input chaining value, and output, without the extra control and changes of the second preimage attack, the received values would not be those that we want, as each side would be affected by the other. But when used with the second preimage attack, with the selected values of the walls, the second preimage attack is able to find a preimage.

Table 3. An Example Preimage of the Compression Function of Snefru (in hexadecimal)

Input chaining value:	00000000	00000000	00000000	00000000	(standard IV)	
Message block:	79F6A75E	0397C368	F60C88DE	3133A55E	6D00251C	8ED3567B
	CA49F82B	A32E5DC4	8F86E479	DD3FF4D6	14DD88C1	A2322E00
Compressed value:	00000000	00000000	00000000	00000000		

We applied this attack on 2-pass Snefru (where the complexity of the second preimage attack is practical), and found many preimages. An example preimage of the compression function of 2-pass Snefru is given in Table 3. Note that this preimage is also a fixpoint of the compression function, and thus it can be easily repeated, without changing the output chaining value.

6.2 Preimage Attacks and the Importance of Padding Schemes

The padding scheme of MD4/SHA optimizes the last block, so that no additional padding block is added unless absolutely necessary (i.e., no extra block is added if the message length and single '1' bit of the padding fit at the end of the last existing block). Thus, the content of the last block can mostly be controlled by the attacker, by selecting messages that are 65-bit short of a multiple of a block. We expect that in such a case, our attack on the compression function would be applicable to the last block, thus leading to a preimage attack on the full hash function (rather than on the compression function only).

However, the padding scheme of Snefru always adds a padding block containing the length only (left filled by '0's). Assuming that the length is bounded by 64 bits, the more restrictive padding scheme of Snefru makes it much more complicated (or maybe impossible using our techniques) to find the preimage of the length block, as the words that the attacker needs to control are fixed to zero. Therefore, the attack on the compression function, as described, cannot work.

It may be the case that an attack can still be found, with a huge complexity (still faster than a generic attack), by changing the locations of the controlled words to the area of the input chaining value and the length, and performing a much more complicated computation, but this would require further research.

Though it looks impossible, we observe that such an attack does not have to follow all the requirements of a preimage attack on the compression function. In particular, such an attack does not have to fix the input chaining value in advance — input chaining value can thus be the output of the attack — making the attack a free-start preimage of the compression function on the length block. This kind of attack may be simpler to find, due to the weaker requirements.

A free-start preimage of the length block suffices for finding a preimage of the full hash function, as after this last block along with it's input chaining value are found, we know the output chaining value required from the previous block. All we need to do at that stage, is to apply the preimage attack on the compression function of that previous block. This situation can be seen in Figure 9, which

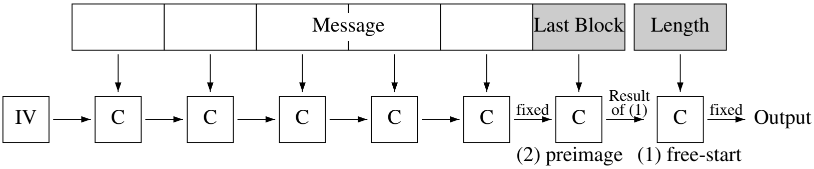


Fig. 9. A free-start preimage of the length block suffices for finding a preimage

describes the mode of operation used by Snefru, where the attack needs to control only the last two blocks (the last block of the message as well as the length block), but in the reverse order (starting from the last block).

6.3 A Preimage of Snefru

We carefully checked the definition of Snefru, and in particular the definition of the padding block. There is no mentioned limit on the message size, as long as the length fits in the last block. Therefore, the attacker can fully control the last block, at the expense of creating messages with a huge number of blocks. There is no difficulty in creating such huge messages, as fixpoints can be found (e.g., the one from Table 3), and iterated as many times as required.

As the length of the block is 384 bits, the length of the iterated message must be a multiple of 384 bits. We would thus need a length block whose content divides by 384. The block of Table 3 is congruent to 256 modulo 384, thus cannot be used as the length block. On the other hand, the preimage of the compression function described in Table 4 represents a multiple of 384, and is also a fixpoint. Therefore, when this block is iterated 79F6A75E CB8E7368 A8532FD9 81175859 CCE2C60C 734D51CF 5E8B7F23 F48893F9 EE56676D 6E565530 9864E5B1 A 2322E00$_x$/384≈ 2^{374} times, it becomes a preimage of the zero hash value! This huge message is a preimage of 2-pass Snefru. By iterating this fixpoint, and using an alternate padding block, it is possible to find preimages for any hash value.

Note that, unfortunately, sending this message would take a huge time. Even verification of this message by the receiver would take a huge time (much more than the preimage attack itself). It would even be faster for the receiver to find another single-block preimage (with complexity 2^{128}), than to receive this message and verify it (which takes about 2^{374} time in the standard verification procedure).

Table 4. The Block Used for the Preimage of Snefru (in hexadecimal)

Input chaining value:	00000000	00000000	00000000	00000000		
Message block:	79F6A75E	CB8E7368	A8532FD9	81175859	CCE2C60C	734D51CF
	5E8B7F23	F48893F9	EE56676D	6E565530	9864E5B1	A2322E00
Compressed value:	00000000	00000000	00000000	00000000		

Table 5. Summary of the Attacks on Snefru

Number of Passes	Second preimage of Compression Function (time)	Preimage of Compression Function (time)	Collision Attack (time)
Novelty:	Old	New	Memoryless
2	2^{24}	2^{32}	$2^{12.5}$
3	2^{56}	2^{56}	$2^{28.5}$
4	2^{88}–2^{112}	2^{88}–2^{112}	$2^{44.5}$–$2^{56.5}$

7 Summary

In this paper, we described new techniques for cryptanalysis of Snefru:

1. A preimage attack based on the second preimage attack with a virtual message.
2. Differential cryptanalysis using generic algorithms.

Table 5 summarizes the complexities of the attacks on Snefru with 2, 3, and 4 passes (and 128-bit hash values). The second column gives the complexities of the second preimage attacks of [2,1]. The third column gives the complexities of the preimage attack on the compression function described in this paper. The last column gives the complexity of the collision attacks of [2,1], which require same size of memory. The memoryless collision attacks described in this paper eliminated the need to that size of memory, without changing the time complexity.

We also discussed the importance of the padding scheme to protect against applying a preimage attack on the compression function for finding preimages of the full hash function.

References

1. Biham, E., Shamir, A.: Differential Cryptanalysis of the Data Encryption Standard. Springer, Heidelberg (1993)
2. Biham, E., Shamir, A.: Differential Cryptanalysis of Snefru, Khafre, REDOC-II, LOKI and Lucifer (extended abstract). In: Feigenbaum, J. (ed.) CRYPTO 1991. LNCS, vol. 576, pp. 156–171. Springer, Heidelberg (1992)
3. Damgård, I.B.: A Design Principle for Hash Functions. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 416–427. Springer, Heidelberg (1990)
4. Knuth, D.E.: The Art of Computer Programming, Seminumerical Algorithms, 3rd edn., vol. 2. Addison-Wesley, Reading (1997)
5. Merkle, R.C.: Secrecy, Authentication, and Public Key Systems. UMI Research press (1982)
6. Merkle, R.C.: One Way Hash Functions and DES. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 428–446. Springer, Heidelberg (1990)
7. Merkle, R.C.: A Fast Software One-Way Hash Function. Journal of Cryptology 3(1), 43–58 (1990)

8. National Institute of Standards and Technology, Secure Hash Standard, U.S. Department of Commerce, FIPS pub. 180-1 (April 1995)
9. Nivasch, G.: Cycle Detection using a Stack. Information Processing Letters 90(3), 135–140 (2004)
10. van Oorschot, P.C., Wiener, M.J.: Parallel Collision Search with Applications to Hash Functions and Discrete Logarithms. In: Proceedings of 2nd ACM Conference on Computer and Communications Security, pp. 210–218. ACM Press, New York (1994)
11. Pollard, J.M.: A Monte Carlo method for factorization. BIT Numerical Mathematics 15(3), 331–334 (1975)
12. Quisquater, J.-J., Delescaille, J.-P.: How Easy is Collision Search? Application to DES. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 429–434. Springer, Heidelberg (1990)
13. Rivest, R.L.: The MD4 Message Digest Algorithm. In: Menezes, A., Vanstone, S.A. (eds.) CRYPTO 1990. LNCS, vol. 537, pp. 303–311. Springer, Heidelberg (1991)
14. Rivest, R.L.: The MD5 Message Digest Algorithm, Internet Request for Comments, RFC 1321 (April 1992)
15. Wang, X., Lai, X., Feng, D., Chen, H., Yu, X.: Cryptanalysis for Hash Functions MD4 and RIPEMD. In: Cramer, R.J.F. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 1–18. Springer, Heidelberg (2005)

On the Salsa20 Core Function

Julio Cesar Hernandez-Castro[1], Juan M.E. Tapiador[2],
and Jean-Jacques Quisquater[1]

[1] Crypto Group, DICE, Universite Louvain-la-Neuve
Place du Levant, 1 B-1348 Louvain-la-Neuve, Belgium
[2] Computer Science Department, Carlos III University
Avda. de la Universidad, 30, 28911 Leganes, Madrid, Spain
julio.hernandez@uclouvain.be, jestevez@inf.uc3m.es,
jjq@uclouvain.be

Abstract. In this paper, we point out some weaknesses in the Salsa20 core function that could be exploited to obtain up to 2^{31} collisions for its full (20 rounds) version. We first find an invariant for its main building block, the **quarterround** function, that is then extended to the **rowround** and **columnround** functions. This allows us to find an input subset of size 2^{32} for which the Salsa20 core behaves exactly as the transformation $f(x) = 2x$. An attacker can take advantage of this for constructing 2^{31} collisions for any number of rounds. We finally show another weakness in the form of a differential characteristic with probability one that proves that the Salsa20 core does not have 2^{nd} preimage resistance.

Keywords: Salsa20, hash function, cryptanalysis, collision.

1 Introduction

Salsa20 is a very interesting design by Daniel Bernstein [1]. It is mostly known because of its submission to the eSTREAM Project, where it passed to Phase 3 without major known attacks, although some interesting weaknesses over reduced-round versions have been pointed out [6,8,11]. As mentioned in [2], *"The core of Salsa20 is a hash function with 64-byte input and 64-byte output. The hash function is used in counter mode as a stream cipher: Salsa20 encrypts a 64-byte block of plaintext by hashing the key, nonce, and block number and xor'ing the result with the plaintext."* Note, however, that in spite of its name, the Salsa20 "hash" function was never really intended for hashing.

Reduced-round versions Salsa20/12 and Salsa20/8 (respectively using 12 and 8 rounds) have been proposed [3], although the author acknowledges that the security margin for Salsa20/8 is not huge, in view of the attack against Salsa20/5 presented in [6]. However, the speed gain over the full Salsa20 is very significant. Unfortunately, serious doubts over the security of Salsa20/8 were raised later over the publication of [11], which essentially breaks Salsa20/6 and successfully attacks Salsa20/7.

K. Nyberg (Ed.): FSE 2008, LNCS 5086, pp. 462–469, 2008.

Salsa20 represents quite an original and flexible design, where the author justifies the use of very simple operations (addition, xor, constant distance rotation) and the lack of multiplication or S-boxes to develop a very fast primitive. Moreover, its construction protects it from timing attacks.

We find that, although this paper shows some vulnerabilities in its underlying cryptographic core, Bernstein's approach is indeed valuable and should be further investigated. For more information about Salsa20 design, please refer to the rationale presented by its author in [4].

The rest of the paper is organized as follows. Section 2 presents the main results in the form of various theorems, and Section 3 shows how these results can be practically used to find collisions for the full Salsa20 "hash" function. Section 4 ends the paper with some conclusions. In the Appendix, we show two collisions (out of the 2^{31} presented in this paper) for testing purposes.

2 Main Results

The main building block of the Salsa20 "hash" is the **quarterround** function, defined as follows:

Definition 1. *If* $y = \begin{pmatrix} y_0 & y_1 \\ y_2 & y_3 \end{pmatrix}$ *then* $\mathbf{quarterround}(y) = \begin{pmatrix} z_0 & z_1 \\ z_2 & z_3 \end{pmatrix}$, *where:*

$$z_1 = y_1 \oplus ((y_0 + y_3) \lll 7) \quad (1)$$

$$z_2 = y_2 \oplus ((z_1 + y_0) \lll 9) \quad (2)$$

$$z_3 = y_3 \oplus ((z_2 + z_1) \lll 13) \quad (3)$$

$$z_0 = y_0 \oplus ((z_3 + z_2) \lll 18) \quad (4)$$

and $X \lll n$ *is the rotation of the 32-bit word* X *to the left by* n *positions.*

Theorem 1. *For any 32-bit value* A*, an input of the form* $\begin{pmatrix} A & -A \\ A & -A \end{pmatrix}$ *is left invariant by the* **quarterround** *function, where* $-A$ *represents the only 32-bit integer satisfying* $A + (-A) = 0 \pmod{2^{32}}$.

Proof. Simply by substituting in the equations above, we obtain that every rotation operates over the null vector, so $z_i = y_i$ for every $i \in (0..3)$ □

Similarly, the **rowround** function, defined below, suffers from the same problem:

Definition 2. *If* $y = \begin{pmatrix} y_0 & y_1 & y_2 & y_3 \\ y_4 & y_5 & y_6 & y_7 \\ y_8 & y_9 & y_{10} & y_{11} \\ y_{12} & y_{13} & y_{14} & y_{15} \end{pmatrix}$ *then* $\mathbf{rowround}(y) = \begin{pmatrix} z_0 & z_1 & z_2 & z_3 \\ z_4 & z_5 & z_6 & z_7 \\ z_8 & z_9 & z_{10} & z_{11} \\ z_{12} & z_{13} & z_{14} & z_{15} \end{pmatrix}$

where:

$$(z_0, z_1, z_2, z_3) = \mathbf{quarterround}(y_0, y_1, y_2, y_3) \quad (5)$$

$$(z_5, z_6, z_7, z_4) = \mathbf{quarterround}(y_5, y_6, y_7, y_4) \quad (6)$$

$$(z_{10}, z_{11}, z_8, z_9) = \textbf{quarterround}(y_{10}, y_{11}, y_8, y_9) \tag{7}$$

$$(z_{15}, z_{12}, z_{13}, z_{14}) = \textbf{quarterround}(y_{15}, y_{12}, y_{13}, y_{14}) \tag{8}$$

Theorem 2. *Any input of the form* $\begin{pmatrix} A & -A & A & -A \\ B & -B & B & -B \\ C & -C & C & -C \\ D & -D & D & -D \end{pmatrix}$*, for any 32-bit values* A, B, C *and* D*, is left invariant by the* ***rowround*** *transformation.*

Proof. This trivially follows from the repeated application of Theorem 1 to the four equations above. □

Remark. It is important to note that any other rearrangement of the equations from its canonical form:

$$(z_{4*i}, z_{4*i+1}, z_{4*i+2}, z_{4*i+3}) = \textbf{quarterround}(y_{4*i}, y_{4*i+1}, y_{4*i+2}, y_{4*i+3}) \tag{9}$$

will suffer from the same problem whenever the rearranging permutation keeps on alternating subindex oddness.

It is worth observing that this result implies that, from the 2^{512} possible inputs, at least one easily characterizable subset of size 2^{128} remains invariant by the **rowround** transformation.

The same happens with the **Columnround** function, which is defined below:

Definition 3. *If* $y = \begin{pmatrix} y_0 & y_1 & y_2 & y_3 \\ y_4 & y_5 & y_6 & y_7 \\ y_8 & y_9 & y_{10} & y_{11} \\ y_{12} & y_{13} & y_{14} & y_{15} \end{pmatrix}$ *then* **columnround**$(y) = \begin{pmatrix} z_0 & z_1 & z_2 & z_3 \\ z_4 & z_5 & z_6 & z_7 \\ z_8 & z_9 & z_{10} & z_{11} \\ z_{12} & z_{13} & z_{14} & z_{15} \end{pmatrix}$ *where:*

$$(z_0, z_4, z_8, z_{12}) = \textbf{quarterround}(y_0, y_4, y_8, y_{12}) \tag{10}$$

$$(z_5, z_9, z_{13}, z_1) = \textbf{quarterround}(y_5, y_9, y_{13}, y_1) \tag{11}$$

$$(z_{10}, z_{14}, z_2, z_6) = \textbf{quarterround}(y_{10}, y_{14}, y_2, y_6) \tag{12}$$

$$(z_{15}, z_3, z_7, z_{11}) = \textbf{quarterround}(y_{15}, y_3, y_7, y_{11}) \tag{13}$$

Theorem 3. *Any input of the form* $\begin{pmatrix} A & -B & C & -D \\ -A & B & -C & D \\ A & -B & C & -D \\ -A & B & -C & D \end{pmatrix}$*, for any 32-bit values* A, B, C *and* D*, is left invariant by the* ***columnround*** *transformation.*

Proof. This follows directly from the repeated application of Theorem 1, and can be seen as a dual of Theorem 2.

Theorem 4. *Any input of the form* $\begin{pmatrix} A & -A & A & -A \\ -A & A & -A & A \\ A & -A & A & -A \\ -A & A & -A & A \end{pmatrix}$ *for any 32-bit value* A*, is left invariant by the* ***doubleround*** *transformation.*

Proof. This is quite obvious. The point is that, due to the arrangement of the indexes in the **columnround** and the **rowround** function, we cannot have as free a hand. Here we are forced to make $B = -A$, $C = A$, and $D = -A$.

Taking into account that **doubleround** is defined as the composition of a **columnround** and a **rowround** operation:

$$\mathbf{doubleround}(x) = \mathbf{rowround}(\mathbf{columnround}(x)) \tag{14}$$

a common fixed point should be also a fixed point of its composition. □

3 Collision Finding for the Salsa20 "Hash" Function

Theorem 5. *For any input of the form* $\begin{pmatrix} A & -A & A & -A \\ -A & A & -A & A \\ A & -A & A & -A \\ -A & A & -A & A \end{pmatrix}$ *and for any 32-bit value A, the Salsa20 core function behaves as a linear transformation of the form* $f(x) = 2x$, *and this happens independently of the number of rounds.*

Proof. As the Salsa20 "hash" is defined as:

$$Salsa20(x) = x + \mathbf{doubleround}^{10}(x) \tag{15}$$

and every input of the above form is an invariant (fixed point) for the **doubleround** function, then:

$$Salsa20(x) = x + \mathbf{doubleround}^{10}(x) = x + x = 2x \tag{16}$$

(And this happens independently of the number of rounds) □

The previous result is of great use in collision finding. All what is left now is to find two different nontrivial inputs, x and x', of the said form such that:

$$x \neq x' \quad \text{but} \quad 2x = 2x' \tag{17}$$

Fortunately, this is possible thanks to modular magic, i.e. the fact that all operations in Salsa20 are performed mod 2^{32}.

3.1 Modular Magic

Let us assume that X is a 32-bit integer such that $X < 2^{31}$. Then, we define $X' = X + 2^{31}$. The interesting point here is that, even though $X \neq X'$, $2X = 2X'$ (mod 2^{32}).

Theorem 6. *Any pair of inputs* $\begin{pmatrix} Z & -Z & Z & -Z \\ -Z & Z & -Z & Z \\ Z & -Z & Z & -Z \\ -Z & Z & -Z & Z \end{pmatrix}$ *and* $\begin{pmatrix} Z' & -Z' & Z' & -Z' \\ -Z' & Z' & -Z' & Z' \\ Z' & -Z' & Z' & -Z' \\ -Z' & Z' & -Z' & Z' \end{pmatrix}$, *such that* $Z < 2^{31}$ *and* $Z' = Z + 2^{31}$, *generate a collision for any number of* *rounds of the Salsa20 "hash" function, producing* $\begin{pmatrix} 2Z & -2Z & 2Z & -2Z \\ -2Z & 2Z & -2Z & 2Z \\ 2Z & -2Z & 2Z & -2Z \\ -2Z & 2Z & -2Z & 2Z \end{pmatrix}$ *as a common hash value.*

Proof. This follows directly from the observations and definitions above. Substitution of the proposed input values into the formulæ for the Salsa20 "hash" will confirm this hypothesis. □

Corollary 1. *Theorem 6 implies that there are at least (these conditions are sufficient but probably not necessary)* 2^{31} *input pairs that generate a collision in the output, proving that indeed Salsa20 is not to be used as-is as a hash function. As an example, two of these pairs are provided in the Appendix.*

Corollary 2. *Let us call inputs of the form discussed by Theorem 5* A-states*. Then, as a direct consequence of Theorem 6 the output by the Salsa20 "hash" function of any* A-state *is also an* A-state *(where, in this case, A is even). It could be interesting to check whether these states could be reached at any intermediate step during a computation beginning with a* non-A state*. This would have important security implications. However, it could be easily shown that this is not the case, so any state leading to an* A-state *should be an* A-state *itself.*

This property has an interesting similitude with Finney-states *for RC4 [7] and could be useful in mounting an impossible fault analysis for the Salsa20 stream cipher, as* Finney-states *were of key importance on the impossible fault cryptanalysis of RC4 [5].* A-states*, on the other hand, have the interesting advantage over* Finney-states *that their influence over the output is immediately recognized, so they can be detected in an even simpler way. On the other hand, it is much less likely to reach an* A-state *by simply injecting random faults, as the set of conditions that should hold is larger than for the RC4 case.*

Once we have shown that the Salsa20 "hash" function is not collision resistant, we focus on its security against 2^{nd} preimage attacks. The next result[1] reveals that 2^{nd} preimage attacks are not only possible but even easy.

Theorem 7. *Any pair of inputs A, B with a difference of*

$$A - B = A \bigoplus B = \begin{pmatrix} \mathit{0x80000000} & \mathit{0x80000000} & \mathit{0x80000000} & \mathit{0x80000000} \\ \mathit{0x80000000} & \mathit{0x80000000} & \mathit{0x80000000} & \mathit{0x80000000} \\ \mathit{0x80000000} & \mathit{0x80000000} & \mathit{0x80000000} & \mathit{0x80000000} \\ \mathit{0x80000000} & \mathit{0x80000000} & \mathit{0x80000000} & \mathit{0x80000000} \end{pmatrix}$$

will produce the same output over any number of rounds.

Proof. This depends on two interesting observations. The first one is that *addition* behaves as *xor* over the most significant bit (that changed by adding `0x80000000`). So the result in each of the four additions on Definition 1 is the same when both its inputs are altered by adding 2^{31} (differences cancel out mod 2^{32}).

[1] This property was presented informally before by Robshaw [10] and later by Wagner [12].

The second one is that in the **quarterround** function, all partial results $z_0, ..., z_3$ are computed after an odd number (three in this case) of *addition/xor* operations. As a result, **quarterround** conserves the input difference, and so it does **rowround**, **columnround** and **doubleround**. As in the last stage of the Salsa20 core function the input is added to the output; This forces input differences to cancel out. □

Corollary 3. *Theorem 7 could now be seen as a particular instance of 6 (because* $2 * \mathit{0x80000000} = \mathit{0x00000000}$*). It is interesting to point out that this result has some common points with the one on the existence of equivalent keys for TEA made by Kelsey et al. [9], and also with the exact truncated differential found by Crowley in [6] for a reduced-round version of the Salsa20 stream cipher.*

A direct consequence of this result is that the effective key/input space of the Salsa20 "hash" is reduced by half, so there is a speed up by a factor of 2 in any exhaustive key/input search attack. This also means that $Salsa20(x) = y$ has solution for no more than (at most) half of the possible y's.

4 Conclusions

The Salsa20 "hash" function was never intended for cryptographic hashing, and some previous results showed that finding a good differential for the core function was not as hard as might have been expected [10]. Even though its author acknowledges that the Salsa20 core is not collision-free, to the best of our knowledge no work has so far focused on finding and characterizing these collisions. In this paper we explicitly show that there is a relevant amount (2^{31}) of easily characterizable collisions, together with an undesirable linear behavior over a large subset of the input space. In a sense, Theorem 6 is a generalization of Robshaw's previous observation.

Since the stream cipher uses four diagonal constants to limit the attacker's control over the input (thus making unreachable the differences needed for a collision), these results have no straightforward implications on its security. However, these undesirable structural properties might be useful to mount an impossible fault attack for the stream cipher. Particularly, what we have called A-states could play a role analogous to Finney states for RC4, in way similar to that presented by Biham et al. at FSE'05 [5]. We consider this as an interesting direction for future research.

That being said, we still consider that Salsa20 design is very innovative and well-motivated. Further work along the same guidelines should be encouraged. Particularly, we believe that a new, perhaps more complex and time consuming definition of the **quarterround** function should lead to a hash that would not be vulnerable to any of the presented attacks and could, in fact, provide a high-level security algorithm. This will, obviously, be more computationally expensive, but there may exist an interesting trade-off between incrementing the complexity of the **quarterround** function and decreasing the total number of rounds. The use of the add-rotate-xor chain at every stage of the **quarterround** function considerably eases the extension of these bad properties to any number of rounds. Although the author justified this approach because of performance reasons, we believe that alternating this structure with xor-rotate-add and making all output words depending on all input words will present the cryptanalyst with a much more difficult task. This should be the subject of further study.

On the other hand, in the light of our results we can also conclude that the inclusion of the diagonal constants is absolutely mandatory. An additional conclusion from our results is that less diagonal constants might suffice for stopping these kinds of undesirable structural properties, with a significant efficiency improvement that can vary from a 16% (from processing 384 bits to 448 bits in the same amount of time, that is, using only two diagonal constants) up to a 33% (in the extreme case of fixing the most significant bit of two diagonal 32-bit values).

Acknowledgments

The authors want to thank the anonymous reviewers, who contributed to improve this paper with their comments and suggestions. We specially want to thank Orr Dunkelman for his insights and many useful remarks.

References

1. Bernstein, D.J.: The Salsa20 Stream Cipher. In: SKEW 2005, Symmetric Key Encryption Workshop, 2005, Workshop Record (2005), http://www.ecrypt.eu.org/stream/salsa20p2.html
2. Bernstein, D.J.: Salsa20 Specification, http://cr.yp.to/snuffle/spec.pdf
3. Bernstein, D.J.: Salsa20/8 and Salsa20/12, http://cr.yp.to/snuffle/812.pdf
4. Bernstein, D.J.: Salsa20 design, http://cr.yp.to/snuffle/design.pdf
5. Biham, E., Granboulan, L., Nguyen, P.Q.: Impossible Fault Analysis of RC4 and Differential Fault Analysis of RC4. In: Gilbert, H., Handschuh, H. (eds.) FSE 2005. LNCS, vol. 3557, pp. 359–367. Springer, Heidelberg (2005)
6. Crowley, P.: Truncated Differential Cryptanalysis of Five Rounds of Salsa20. In: eSTREAM, ECRYPT Stream Cipher Project, Report 2005/073
7. Finney, H.: An RC4 Cycle that Cant Happen. sci.crypt newsgroup (September 1994)
8. Fischer, S., Meier, W., Berbain, C., Biasse, J.-F., Robshaw, M.: Non-Randomness in eSTREAM Candidates Salsa20 and TSC-4. In: Barua, R., Lange, T. (eds.) INDOCRYPT 2006. LNCS, vol. 4329, pp. 2–16. Springer, Heidelberg (2006)
9. Kelsey, J., Schneier, B., Wagner, D.: Key-schedule cryptanalysis of IDEA, G-DES, GOST, SAFER, and Triple-DES. In: Koblitz, N. (ed.) CRYPTO 1996. LNCS, vol. 1109, pp. 237–251. Springer, Heidelberg (1996)
10. Robshaw, M.: The Salsa20 Hash Function is Not Collision-Free June 22 (2005)
11. Tsunoo, Y., Saito, T., Kubo, H., Suzaki, T., Nakashima, H.: Differential Cryptanalysis of Salsa20/8 (submitted, 2007-01-02), http://www.ecrypt.eu.org/stream/papersdir/2007/010.pdf
12. Wagner, D.: Message from discussion "Re-rolled Salsa-20 function" in the sci.crypt newsgroup on September 26th (2005), http://groups.google.com/group/sci.crypt/msg/0692e3aaf78687a3

Appendix: Collisions for the Full Salsa20 Hash Function

Here we show a couple of collisions for testing purposes:

If

$$Z = \begin{pmatrix} \texttt{0xAAAAAAAA} & \texttt{0x55555556} & \texttt{0xAAAAAAAA} & \texttt{0x55555556} \\ \texttt{0x55555556} & \texttt{0xAAAAAAAA} & \texttt{0x55555556} & \texttt{0xAAAAAAAA} \\ \texttt{0xAAAAAAAA} & \texttt{0x55555556} & \texttt{0xAAAAAAAA} & \texttt{0x55555556} \\ \texttt{0x55555556} & \texttt{0xAAAAAAAA} & \texttt{0x55555556} & \texttt{0xAAAAAAAA} \end{pmatrix}$$

and

$$Z' = \begin{pmatrix} \texttt{0x2AAAAAAA} & \texttt{0xD5555556} & \texttt{0x2AAAAAAA} & \texttt{0xD5555556} \\ \texttt{0xD5555556} & \texttt{0x2AAAAAAA} & \texttt{0xD5555556} & \texttt{0x2AAAAAAA} \\ \texttt{0x2AAAAAAA} & \texttt{0xD5555556} & \texttt{0x2AAAAAAA} & \texttt{0xD5555556} \\ \texttt{0xD5555556} & \texttt{0x2AAAAAAA} & \texttt{0xD5555556} & \texttt{0x2AAAAAAA} \end{pmatrix}$$

then, the common Salsa20 hash value is

$$Salsa20(Z)=Salsa20(Z')=\begin{pmatrix} \texttt{0x55555554} & \texttt{0xAAAAAAAC} & \texttt{0x55555554} & \texttt{0xAAAAAAAC} \\ \texttt{0xAAAAAAAC} & \texttt{0x55555554} & \texttt{0xAAAAAAAC} & \texttt{0x55555554} \\ \texttt{0x55555554} & \texttt{0xAAAAAAAC} & \texttt{0x55555554} & \texttt{0xAAAAAAAC} \\ \texttt{0xAAAAAAAC} & \texttt{0x55555554} & \texttt{0xAAAAAAAC} & \texttt{0x55555554} \end{pmatrix}$$

Alternatively, if

$$W = \begin{pmatrix} \texttt{0xFFFFFFFF} & \texttt{0x00000001} & \texttt{0xFFFFFFFF} & \texttt{0x00000001} \\ \texttt{0x00000001} & \texttt{0xFFFFFFFF} & \texttt{0x00000001} & \texttt{0xFFFFFFFF} \\ \texttt{0xFFFFFFFF} & \texttt{0x00000001} & \texttt{0xFFFFFFFF} & \texttt{0x00000001} \\ \texttt{0x00000001} & \texttt{0xFFFFFFFF} & \texttt{0x00000001} & \texttt{0xFFFFFFFF} \end{pmatrix}$$

and

$$W' = \begin{pmatrix} \texttt{0x7FFFFFFF} & \texttt{0x80000001} & \texttt{0x7FFFFFFF} & \texttt{0x80000001} \\ \texttt{0x80000001} & \texttt{0x7FFFFFFF} & \texttt{0x80000001} & \texttt{0x7FFFFFFF} \\ \texttt{0x7FFFFFFF} & \texttt{0x80000001} & \texttt{0x7FFFFFFF} & \texttt{0x80000001} \\ \texttt{0x80000001} & \texttt{0x7FFFFFFF} & \texttt{0x80000001} & \texttt{0x7FFFFFFF} \end{pmatrix}$$

then, the common Salsa20 hash value is

$$Salsa20(W)=Salsa20(W')=\begin{pmatrix} \texttt{0xFFFFFFFE} & \texttt{0x00000002} & \texttt{0xFFFFFFFE} & \texttt{0x00000002} \\ \texttt{0x00000002} & \texttt{0xFFFFFFFE} & \texttt{0x00000002} & \texttt{0xFFFFFFFE} \\ \texttt{0xFFFFFFFE} & \texttt{0x00000002} & \texttt{0xFFFFFFFE} & \texttt{0x00000002} \\ \texttt{0x00000002} & \texttt{0xFFFFFFFE} & \texttt{0x00000002} & \texttt{0xFFFFFFFE} \end{pmatrix}$$

New Features of Latin Dances: Analysis of Salsa, ChaCha, and Rumba

Jean-Philippe Aumasson[1], Simon Fischer[1], Shahram Khazaei[2], Willi Meier[1], and Christian Rechberger[3]

[1] FHNW, Windisch, Switzerland
[2] EPFL, Lausanne, Switzerland
[3] IAIK, Graz, Austria

Abstract. The stream cipher Salsa20 was introduced by Bernstein in 2005 as a candidate in the eSTREAM project, accompanied by the reduced versions Salsa20/8 and Salsa20/12. ChaCha is a variant of Salsa20 aiming at bringing better diffusion for similar performance. Variants of Salsa20 with up to 7 rounds (instead of 20) have been broken by differential cryptanalysis, while ChaCha has not been analyzed yet. We introduce a novel method for differential cryptanalysis of Salsa20 and ChaCha, inspired by correlation attacks and related to the notion of neutral bits. This is the first application of neutral bits in stream cipher cryptanalysis. It allows us to break the 256-bit version of Salsa20/8, to bring faster attacks on the 7-round variant, and to break 6- and 7-round ChaCha. In a second part, we analyze the compression function Rumba, built as the XOR of four Salsa20 instances and returning a 512-bit output. We find collision and preimage attacks for two simplified variants, then we discuss differential attacks on the original version, and exploit a high-probability differential to reduce complexity of collision search from 2^{256} to 2^{79} for 3-round Rumba. To prove the correctness of our approach we provide examples of collisions and near-collisions on simplified versions.

1 Introduction

Salsa20 [5] is a stream cipher introduced by Bernstein in 2005 as a candidate in the eSTREAM project [12], that has been selected in April 2007 for the third and ultimate phase of the competition. Three independent cryptanalyses were published [11,13,16], reporting key-recovery attacks for reduced versions with up to 7 rounds, while Salsa20 has a total of 20 rounds. Bernstein also submitted to public evaluation the 8- and 12-round variants Salsa20/8 and Salsa20/12 [6], though they are not formal eSTREAM candidates. Later he introduced ChaCha [4,3,8], a variant of Salsa20 that aims at bringing faster diffusion without slowing down encryption.

The compression function Rumba [7] was presented in 2007 in the context of a study of generalized birthday attacks [17] applied to incremental hashing [2], as the component of a hypothetical iterated hashing scheme. Rumba maps a 1536-bit value to a 512-bit (intermediate) digest, and Bernstein only conjectures collision resistance for this function, letting a further convenient operating mode provide extra security properties as pseudo-randomness.

K. Nyberg (Ed.): FSE 2008, LNCS 5086, pp. 470–488, 2008.

Related Work. Variants of Salsa20 up to 7 rounds have been broken by differential cryptanalysis, exploiting a truncated differential over 3 or 4 rounds. The knowledge of less than 256 key bits can be sufficient for observing a difference in the state after three or four rounds, given a block of keystream of up to seven rounds of Salsa20. In 2005, Crowley [11] reported a 3-round differential, and built upon this an attack on Salsa20/5 within claimed 2^{165} trials. In 2006, Fischer et al. [13] exploited a 4-round differential to attack Salsa20/6 within claimed 2^{177} trials. In 2007, Tsunoo et al. [16] attacked Salsa20/7 within about 2^{190} trials, still exploiting a 4-round differential, and also claimed a break of Salsa20/8. However, the latter attack is effectively slower than brute force, cf. §3.5. Tsunoo et al. notably improve from previous attacks by reducing the guesses to certain bits—rather than guessing whole key words—using nonlinear approximation of integer addition. Eventually, no attack on ChaCha or Rumba has been published so far.

Contribution. We introduce a novel method for attacking Salsa20 and ChaCha (and potentially other ciphers) inspired from correlation attacks, and from the notion of neutral bit, introduced by Biham and Chen [9] for attacking SHA-0. More precisely, we use an empirical measure of the correlation between certain key bits of the state and the bias observed after working a few rounds backward, in order to split key bits into two subsets: the extremely relevant key bits to be subjected to an exhaustive search and filtered by observations of a biased output-difference value,and the less significant key bits ultimately determined by exhaustive search. To the best of our knowledge, this is the first time that neutral bits are used for the analysis of stream ciphers. Our results are summarized in Tab. 1. We present the first key-recovery attack for the 256-bit version of Salsa20/8, improve the previous attack on 7-round Salsa20 by a factor 2^{39}, and present attacks on ChaCha up to 7 rounds. The 128-bit versions are also investigated. In a second part, we first show collision and preimage attacks for simplified versions of Rumba, then we present a differential analysis of the original version using the methods of linearization and neutral bits: our main result is a collision attack for 3-round Rumba running in about 2^{79} trials (compared to 2^{256} with a birthday attack). We also give examples of near-collisions over three and four rounds.

Table 1. Complexity of the best attacks known, with success probability 1/2

	Salsa20/7	Salsa20/8	ChaCha6	ChaCha7	Rumba3
Before	2^{190}	2^{255}	2^{255}	2^{255}	2^{256}
Now	2^{151}	2^{251}	2^{139}	2^{248}	2^{79}

Road Map. We first recall the definitions of Salsa20, ChaCha, and Rumba in §2, then §3 describes our attacks on Salsa20 and ChaCha, and §4 presents our cryptanalysis of Rumba. The appendices give the sets of constant values, and some parameters necessary to reproduce our attacks.

2 Specification of Primitives

In this section, we give a concise description of the stream ciphers Salsa20 and ChaCha, and of the compression function Rumba.

2.1 Salsa20

The stream cipher Salsa20 operates on 32-bit words, takes as input a 256-bit key $k = (k_0, k_1, \ldots, k_7)$ and a 64-bit nonce $v = (v_0, v_1)$, and produces a sequence of 512-bit keystream blocks. The i-th block is the output of the *Salsa20 function*, that takes as input the key, the nonce, and a 64-bit counter $t = (t_0, t_1)$ corresponding to the integer i. This function acts on the 4×4 matrix of 32-bit words written as

$$X = \begin{pmatrix} x_0 & x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 & x_7 \\ x_8 & x_9 & x_{10} & x_{11} \\ x_{12} & x_{13} & x_{14} & x_{15} \end{pmatrix} = \begin{pmatrix} c_0 & k_0 & k_1 & k_2 \\ k_3 & c_1 & v_0 & v_1 \\ t_0 & t_1 & c_2 & k_4 \\ k_5 & k_6 & k_7 & c_3 \end{pmatrix} . \tag{1}$$

The c_i's are predefined constants (see Appendix A). There is also a mode for a 128-bit key k', where the 256 key bits in the matrix are filled with $k = k' \| k'$. If not mentioned otherwise, we focus on the 256-bit version. A keystream block Z is then defined as

$$Z = X + X^{20} , \tag{2}$$

where "+" symbolizes wordwise integer addition, and where $X^r = \mathsf{Round}^r(X)$ with the round function Round of Salsa20. The round function is based on the following nonlinear operation (also called the $\mathsf{quarterround}$ function), which transforms a vector (x_0, x_1, x_2, x_3) to (z_0, z_1, z_2, z_3) by sequentially computing

$$\begin{aligned} z_1 &= x_1 \oplus \big[(x_3 + x_0) \lll 7\ \big] \\ z_2 &= x_2 \oplus \big[(x_0 + z_1) \lll 9\ \big] \\ z_3 &= x_3 \oplus \big[(z_1 + z_2) \lll 13\big] \\ z_0 &= x_0 \oplus \big[(z_2 + z_3) \lll 18\big] . \end{aligned} \tag{3}$$

In odd numbers of rounds (which are called $\mathsf{columnrounds}$ in the original specification of Salsa20), the nonlinear operation is applied to the columns (x_0, x_4, x_8, x_{12}), (x_5, x_9, x_{13}, x_1), $(x_{10}, x_{14}, x_2, x_6)$, $(x_{15}, x_3, x_7, x_{11})$. In even numbers of rounds (which are also called the $\mathsf{rowrounds}$), the nonlinear operation is applied to the rows (x_0, x_1, x_2, x_3), (x_5, x_6, x_7, x_4), $(x_{10}, x_{11}, x_8, x_9)$, $(x_{15}, x_{12}, x_{13}, x_{14})$. We write Salsa20/R for R-round variants, i.e. with $Z = X + X^R$. Note that the r-round inverse $X^{-r} = \mathsf{Round}^{-r}(X)$ is defined differently whether it inverts after an odd or and even number of rounds.

2.2 ChaCha

ChaCha is similar to Salsa20 with the following modifications

1. The input words are placed differently in the initial matrix:

$$X = \begin{pmatrix} x_0 & x_1 & x_2 & x_3 \\ x_4 & x_5 & x_6 & x_7 \\ x_8 & x_9 & x_{10} & x_{11} \\ x_{12} & x_{13} & x_{14} & x_{15} \, . \end{pmatrix} = \begin{pmatrix} c_0 & c_1 & c_2 & c_3 \\ k_0 & k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 & k_7 \\ t_0 & t_1 & v_0 & v_1 \end{pmatrix} \, . \tag{4}$$

2. The nonlinear operation of Round transforms a vector (x_0, x_1, x_2, x_3) to (z_0, z_1, z_2, z_3) by sequentially computing

$$\begin{aligned} b_0 &= x_0 + x_1, & b_3 &= (x_3 \oplus b_0) \lll 16 \\ b_2 &= x_2 + b_3, & b_1 &= (x_1 \oplus b_2) \lll 12 \\ z_0 &= b_0 + b_1, & z_3 &= (b_3 \oplus z_0) \lll 8 \\ z_2 &= b_2 + z_3, & z_1 &= (b_1 \oplus z_2) \lll 7 \, . \end{aligned} \tag{5}$$

3. The round function is defined differently: in odd numbers of rounds, the nonlinear operation is applied to the columns (x_0, x_4, x_8, x_{12}), (x_1, x_5, x_9, x_{13}), $(x_2, x_6, x_{10}, x_{14})$, $(x_3, x_7, x_{11}, x_{15})$, and in even numbers of rounds, the nonlinear operation is applied to the diagonals $(x_0, x_5, x_{10}, x_{15})$, $(x_1, x_6, x_{11}, x_{12})$, (x_2, x_7, x_8, x_{13}), (x_3, x_4, x_9, x_{14}), see [3] for details.

As for Salsa20, the round function of ChaCha is trivially invertible. R-round variants are denoted ChaChaR. The core function of ChaCha suggests that *"the big advantage of ChaCha over Salsa20 is the diffusion, which at least at first glance looks considerably faster"* [4].

2.3 Rumba

Rumba is a *compression function* built on Salsa20, mapping a 1536-bit message to a 512-bit value. The input M is parsed as four 384-bit chunks $M_0,\ldots,M_3$, and Rumba's output is

$$\begin{aligned} \text{Rumba}(M) &= F_0(M_0) \oplus F_1(M_1) \oplus F_2(M_2) \oplus F_3(M_3) \\ &= (X_0 + X_0^{20}) \oplus (X_1 + X_1^{20}) \oplus (X_2 + X_2^{20}) \oplus (X_3 + X_3^{20}) \, , \end{aligned} \tag{6}$$

where each F_i is an instance of the function Salsa20 with distinct diagonal constants (see Appendix A). The 384-bit input chunk M_i along with the corresponding 128-bit diagonal constants are then used to fill up the corresponding input matrix X_i. A single word j of X_i is denoted $x_{i,j}$. Note that the functions F_i include the feedforward operation of Salsa20. RumbaR stands for R-round variant.

3 Differential Analysis of Salsa20 and ChaCha

This section introduces differential attacks based on a new technique called *probabilistic neutral bits* (shortcut PNB's). To apply it to Salsa20 and ChaCha, we first identify optimal choices of truncated differentials, then we describe a general framework for probabilistic backwards computation, and introduce the notion of PNB's along with a method to find them. Then, we outline the overall attack, and present concrete attacks for Salsa20/7, Salsa20/8, ChaCha6, and ChaCha7. Eventually, we discuss our attack scenarios and possibilities of improvements.

3.1 Choosing a Differential

Let x_i be the i-th word of the matrix-state X, and x'_i an associated word with the difference $\Delta_i^0 = x_i \oplus x'_i$. The j-th bit of x_i is denoted $[x_i]_j$. We use (truncated) input/output differentials for the input X, with a single-bit input-difference $[\Delta_i^0]_j = 1$ in the nonce, and consider a single-bit output-difference $[\Delta_p^r]_q$ after r rounds in X^r. Such a differential is denoted $([\Delta_p^r]_q \mid [\Delta_i^0]_j)$. For a fixed key, the bias ε_d of the output-difference is defined by

$$\Pr_{v,t}\{[\Delta_p^r]_q = 1 \mid [\Delta_i^0]_j\} = \frac{1}{2}(1 + \varepsilon_d) \ , \tag{7}$$

where the probability holds over all nonces and counters. Furthermore, considering key as a random variable, we denote the median value of of ε_d by $\varepsilon_d^\star$. Hence, for half of the keys this differential will have a bias of at least $\varepsilon_d^\star$. Note that our statistical model considers a (uniformly) random value of the counter. In the following, we use the shortcuts $\mathcal{ID}$ and $\mathcal{OD}$ for input- and output-difference.

3.2 Probabilistic Backwards Computation

In the following, assume that the differential $([\Delta_p^r]_q \mid [\Delta_i^0]_j)$ of bias ε_d is fixed, and the corresponding outputs Z and Z' are observed for nonce v, counter t and key k. Having k, v and t, one can invert the operations in $Z = X + X^R$ and $Z' = X' + (X')^R$ in order to access to the r-round forward differential (with $r < R$) from the backward direction thanks to the relations $X^r = (Z - X)^{r-R}$ and $(X')^r = (Z' - X')^{r-R}$. More specifically, define $f(k, v, t, Z, Z')$ as the function which returns the q-th LSB of the word number p of the matrix $(Z - X)^{r-R} \oplus (Z' - X')^{r-R}$, hence $f(k, v, t, Z, Z') = [\Delta_p^r]_q$. Given enough output block pairs with the presumed difference in the input, one can verify the correctness of a guessed candidate $\hat{k}$ for the key k by evaluating the bias of the function f. More precisely, we have $\Pr\{f(\hat{k}, v, t, Z, Z') = 1\} = \frac{1}{2}(1 + \varepsilon_d)$ conditioned on $\hat{k} = k$, whereas for (almost all) $\hat{k} \neq k$ we expect f be unbiased i.e. $\Pr\{f(\hat{k}, v, t, Z, Z') = 1\} = \frac{1}{2}$. The classical way of finding the correct key requires exhaustive search over all possible 2^{256} guesses $\hat{k}$. However, we can search only over a subkey of $m = 256 - n$ bits, provided that an approximation g of f which effectively depends on m key bits is available. More formally, let $\bar{k}$ correspond to the subkey of m bits of the key k and let f be correlated to g with bias ε_a i.e.:

$$\Pr_{v,t}\{f(k, v, t, Z, Z') = g(\bar{k}, v, t, Z, Z')\} = \frac{1}{2}(1 + \varepsilon_a) \ . \tag{8}$$

Note that deterministic backwards computation (i.e. $\bar{k} = k$ with $f = g$) is a special case with $\varepsilon_a = 1$. Denote the bias of g by ε, i.e. $\Pr\{g(\bar{k}, v, t, Z, Z') = 1\} = \frac{1}{2}(1 + \varepsilon)$. Under some reasonable independency assumptions, the equality $\varepsilon = \varepsilon_d \cdot \varepsilon_a$ holds. Again, we denote $\varepsilon^\star$ the median bias over all keys (we verified in experiments that $\varepsilon^\star$ can be well estimated by the median of $\varepsilon_d \cdot \varepsilon_a$). Here, one can verify the correctness of a guessed candidate $\hat{\bar{k}}$ for the subkey $\bar{k}$ by evaluating the bias of the function g based on the fact that we have $\Pr\{g(\hat{\bar{k}}, v, t, Z, Z') = 1\} = \frac{1}{2}(1 + \varepsilon)$ for $\hat{\bar{k}} = \bar{k}$, whereas $\Pr\{g(\hat{\bar{k}}, v, t, Z, Z') = 1\} = \frac{1}{2}$ for $\hat{\bar{k}} \neq \bar{k}$. This way we are facing an exhaustive search over 2^m subkey candidates opposed to the original 2^{256} key candidates which can potentially lead to a faster attack. We stress that the price which we pay is a higher data complexity, see §3.4 for more details.

3.3 Probabilistic Neutral Bits

Our new view of the problem, described in §3.2, demands efficient ways for finding suitable approximations $g(\bar{k}, W)$ of a given function $f(k, W)$ where W is a known parameter; in our case, it is $W = (v, t, Z, Z')$. In a probabilistic model one can consider W as a uniformly distributed random variable. Finding such approximations in general is an interesting open problem. In this section we introduce a generalized concept of neutral bits [9] called *probabilistic neutral bits* (PNB's). This will help us to find suitable approximations in the case that the Boolean function f does not properly mix its input bits. Generally speaking, PNB's allows us to divide the key bits into two groups: *significant key bits* (of size m) and *non-significant key bits* (of size n). In order to identify these two sets we focus on the amount of influence which each bit of the key has on the output of f. Here is a formal definition of a suitable measure:

Definition 1. *The neutrality measure of the key bit k_i with respect to the function $f(k, W)$ is defined as γ_i, where $\Pr = \frac{1}{2}(1 + \gamma_i)$ is the probability (over all k and W) that complementing the key bit k_i does not change the output of $f(k, W)$.*

Singular cases of the neutrality measure are:

- $\gamma_i = 1$: $f(k, W)$ does not depend on i-th key bit (i.e. it is a neutral bit).
- $\gamma_i = 0$: $f(k, W)$ is statistically independent of the i-th key bit (i.e. it is a significant bit).
- $\gamma_i = -1$: $f(k, W)$ linearly depends on the i-th key bit.

In practice, we set a threshold γ and put all key bits with $\gamma_i \leq \gamma$ in the set of significant key bits. The less significant key bits we get, the faster the attack will be, provided that the bias ε_a (see Eq. 8) remains non-negligible. Having found significant and non-significant key bits, we simply let $\bar{k}$ be the significant key bits and define $g(\bar{k}, W)$ as $f(k, W)$ with non-significant key bits being set to a fixed value (e.g. all zero). Note that, contrary to the mutual interaction between neutral bits in [9], here we have directly combined several PNB's without altering their probabilistic quality. This can be justified as the bias ε_a smoothly decreases while we increase the threshold γ.

Remark 1. Tsunoo et al. [16] used nonlinear approximations of integer addition to identify the dependency of key bits, whereas the independent key bits—with respect to nonlinear approximation of some order—are fixed. This can be seen as a special case of our method.

3.4 Complexity Estimation

Here we sketch the full attack described in the previous subsections, then study its computational cost. The attack is split up into a precomputation stage, and a stage of effective attack; note that precomputation is not specific to a key or a counter.

Precomputation

1. Find a high-probability r-round differential with $\mathcal{ID}$ in the nonce or counter.
2. Choose a threshold γ.
3. Construct the function f defined in §3.2.
4. Empirically estimate the neutrality measure γ_i of each key bit for f.
5. Put all those key bits with $\gamma_i < \gamma$ in the significant key bits set (of size $m = 256 - n$).
6. Construct the function g using f by assigning a fixed value to the non-significant key bits, see §3.2 and §3.3.
7. Estimate the median bias $\varepsilon^\star$ by empirically measuring the bias of g using many randomly chosen keys, see §3.2.
8. Estimate the data and time complexity of the attack, see the following.

The cost of this precomputation phase is negligible compared to the effective attack (to be explained later). The r-round differential and the threshold γ should be chosen such that the resulting time complexity is optimal. This will be addressed later in this section. At step 1, we require the difference to be in the nonce or in the counter, assuming that both variables are user-controlled inputs. We exclude a difference in the key in a *related-key* attack due to the disputable attack model. Previous attacks on Salsa20 use the rough estimate of $N = \varepsilon^{-2}$ samples, in order to identify the correct subkey in a large search space. However this estimate is incorrect: this is the number of samples necessary to identify a *single* random unknown bit from either a uniform source or from a non-uniform source with ε, which is a different problem of hypothesis testing. In our case, we have a set of 2^m sequences of random variables with $2^m - 1$ of them verifying the null hypothesis H_0, and a single one verifying the alternative hypothesis H_1. For a realization a of the corresponding random variable A, the decision rule $\mathcal{D}(a) = i$ to accept H_i can lead to two types of errors:

1. Non-detection: $\mathcal{D}(a) = 0$ and $A \in H_1$. The probability of this event is p_{nd}.
2. False alarm: $\mathcal{D}(a) = 1$ and $A \in H_0$. The probability of this event is p_{fa}.

The Neyman-Pearson decision theory gives results to estimate the number of samples N required to get some bounds on the probabilities. It can be shown that

$$N \approx \left(\frac{\sqrt{\alpha \log 4} + 3\sqrt{1 - \varepsilon^2}}{\varepsilon} \right)^2 \tag{9}$$

samples suffices to achieve $p_{\text{nd}} = 1.3 \times 10^{-3}$ and $p_{\text{fa}} = 2^{-\alpha}$. Calculus details and the construction of the optimal distinguisher can be found in [15], see also [1] for more general results on distributions' distinguishability. In our case the value of ε is key dependent, so we use the median bias $\varepsilon^\star$ in place of ε in Eq. 9, resulting in a success probability of at least $\frac{1}{2}(1-p_{\mathrm{nd}}) \approx \frac{1}{2}$ for our attack. Having determined the required number of samples N and the optimal distinguisher, we can now present the effective (or online) attack.

Effective attack

1. For an unknown key, collect N pairs of keystream blocks where each pair is produced by states with a random nonce and counter (satisfying the relevant $\mathcal{ID}$).
2. For each choice of the subkey (i.e. the m significant key bits) do:
   (a) Compute the bias of g using the N keystream block pairs.
   (b) If the optimal distinguisher legitimates the subkeys candidate as a (possibly) correct one, perform an additional exhaustive search over the n non-significant key bits in order to check the correctness of this filtered subkey and to find the non-significant key bits.
   (c) Stop if the right key is found, and output the recovered key.

Let us now discuss the time complexity of our attack. Step 2 is repeated for all 2^m subkey candidates. For each subkey, step (a) is always executed which has complexity[1] of N. However, the search part of step (b) is performed only with probability $p_{\mathrm{fa}} = 2^{-\alpha}$ which brings an additional cost of 2^n in case a subkey passes the optimal distinguisher's filter. Therefore the complexity of step (b) is $2^n p_{\mathrm{fa}}$, showing a total complexity of $2^m(N + 2^n p_{\mathrm{fa}}) = 2^m N + 2^{256-\alpha}$ for the effective attack. In practice, α (and hence N) is chosen such that it minimizes $2^m N + 2^{256-\alpha}$. Note that the potential improvement from key ranking techniques is not considered here, see e.g. [14]. The data complexity of our attack is N keystream block pairs.

Remark 2. It is reasonable to assume that a false subkey, which is close to the correct subkey, may introduce a non-negligible bias. In general, this results in an increased value of p_{fa}. If many significant key bits have neutrality measure close to zero, then the increase is expected to be small, but the precise practical impact of this observation is unknown to the authors.

3.5 Experimental Results

We used automatized search to identify optimal differentials for the reduced-round versions Salsa20/7, Salsa20/8, ChaCha6, and ChaCha7. This search is based on the following observation: The number n of PNB's for some fixed threshold γ mostly depends on the $\mathcal{OD}$, but not on the $\mathcal{ID}$. Consequently, for each of the 512 single-bit $\mathcal{OD}$'s, we can assign the $\mathcal{ID}$ with maximum bias ε_d, and estimate time complexity of the attack. Below we only present the differentials leading to the best attacks. The threshold γ is also an important parameter:

[1] More precisely the complexity is about $2(R-r)/RN$ times the required time for producing one keystream block.

Given a fixed differential, time complexity of the attack is minimal for some optimal value of γ. However, this optimum may be reached for quite small γ, such that n is large and $|\varepsilon_a^\star|$ small. We use at most 2^{24} random nonces and counters for each of the 2^{10} random keys, so we can only measure a bias of about $|\varepsilon_a^\star| > c \cdot 2^{-12}$ (where $c \approx 10$ for a reasonable estimation error). In our experiments, the optimum is not reached with these computational possibilities (see e.g. Tab. 2), and we note that the described complexities may be improved by choosing a smaller γ.

Attack on 256-bit Salsa20/7. We use the differential $([\Delta_1^4]_{14} \mid [\Delta_7^0]_{31})$ with $|\varepsilon_d^\star| = 0.131$. The $\mathcal{OD}$ is observed after working three rounds backward from a 7-round keystream block. To illustrate the role of the threshold γ, we present in Tab. 2 complexity estimates along with the number n of PNB's, the values of $|\varepsilon_d^\star|$ and $|\varepsilon^\star|$, and the optimal values of α for several threshold values. For $\gamma = 0.4$, the attack runs in time 2^{151} and data 2^{26}. The previous best attack in [16] required about 2^{190} trials and 2^{12} data.

Table 2. Different parameters for our attack on 256-bit Salsa20/7

γ	n	$\lvert\varepsilon_a^\star\rvert$	$\lvert\varepsilon^\star\rvert$	α	Time	Data
1.00	39	1.000	0.1310	31	2^{230}	2^{13}
0.90	97	0.655	0.0860	88	2^{174}	2^{15}
0.80	103	0.482	0.0634	93	2^{169}	2^{16}
0.70	113	0.202	0.0265	101	2^{162}	2^{19}
0.60	124	0.049	0.0064	108	2^{155}	2^{23}
0.50	131	0.017	0.0022	112	2^{151}	2^{26}

Attack on 256-bit Salsa20/8. We use again the differential $([\Delta_1^4]_{14} \mid [\Delta_7^0]_{31})$ with $|\varepsilon_d^\star| = 0.131$. The $\mathcal{OD}$ is observed after working four rounds backward from an 8-round keystream block. For the threshold $\gamma = 0.12$ we find $n = 36$, $|\varepsilon_a^\star| = 0.0011$, and $|\varepsilon^\star| = 0.00015$. For $\alpha = 8$, this results in time 2^{251} and data 2^{31}. The list of PNB's is {26, 27, 28, 29, 30, 31, 71, 72, 120, 121, 122, 148, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 210, 211, 212, 224, 225, 242, 243, 244, 245, 246, 247}. Note that our attack reaches the same success probability and supports an identical degree of parallelism as brute force. The previous attack in [16] claims 2^{255} trials with data 2^{10} for success probability 44%, but exhaustive search succeeds with probability 50% within the same number of trials, with much less data and no additional computations. Therefore their attack does not constitute a break of Salsa20/8.

Attack on 128-bit Salsa20/7. Our attack can be adapted to the 128-bit version of Salsa20/7. With the differential $([\Delta_1^4]_{14} \mid [\Delta_7^0]_{31})$ and $\gamma = 0.4$, we find $n = 38$, $|\varepsilon_a^\star| = 0.045$, and $|\varepsilon^\star| = 0.0059$. For $\alpha = 21$, this breaks Salsa20/7 within 2^{111} time and 2^{21} data. Our attack fails to break 128-bit Salsa20/8 because of the insufficient number of PNB's.

Attack on 256-bit ChaCha6. We use the differential $([\Delta^3_{11}]_0 \mid [\Delta^0_{13}]_{13})$ with $|\varepsilon^\star_d| = 0.026$. The $\mathcal{OD}$ is observed after working three rounds backward from an 6-round keystream block. For the threshold $\gamma = 0.6$ we find $n = 147$, $|\varepsilon^\star_a| = 0.018$, and $|\varepsilon^\star| = 0.00048$. For $\alpha = 123$, this results in time 2^{139} and data 2^{30}.

Attack on 256-bit ChaCha7. We use again the differential $([\Delta^3_{11}]_0 \mid [\Delta^0_{13}]_{13})$ with $|\varepsilon^\star_d| = 0.026$. The $\mathcal{OD}$ is observed after working four rounds backward from an 7-round keystream block. For the threshold $\gamma = 0.5$ we find $n = 35$, $|\varepsilon^\star_a| = 0.023$, and $|\varepsilon^\star| = 0.00059$. For $\alpha = 11$, this results in time 2^{248} and data 2^{27}. The list of PNB's is {3, 6, 15, 16, 31, 35, 67, 68, 71, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 103, 104, 127, 136, 191, 223, 224, 225, 248, 249, 250, 251, 252, 253, 254, 255}.

Attack on 128-bit ChaCha6. Our attack can be adapted to the 128-bit version of ChaCha6. With the differential $([\Delta^3_{11}]_0 \mid [\Delta^0_{13}]_{13})$ and $\gamma = 0.5$, we find $n = 51$, $|\varepsilon^\star_a| = 0.013$, and $|\varepsilon^\star| = 0.00036$. For $\alpha = 26$, this breaks ChaCha6 within 2^{107} time and 2^{30} data. Our attack fails to break 128-bit ChaCha7.

3.6 Discussion

Our attack on reduced-round 256-bit Salsa20 exploits a 4-round differential, to break the 8-round cipher by working four rounds backward. For ChaCha, we use a 3-round differential to break 7 rounds. We made intensive experiments for observing a bias after going five rounds backwards from the guess of a subkey, in order to attack Salsa20/9 or ChaCha8, but without success. Four seems to be the highest number of rounds one can invert from a partial key guess, while still observing a non-negligible bias after inversion, and such that the overall cost improves from exhaustive key search. Can one hope to break further rounds by statistical cryptanalysis? We believe that it would require novel techniques and ideas, rather than the relatively simple XOR difference of 1-bit input and 1-bit output. For example, one might combine several biased $\mathcal{OD}$'s to reduce the data complexity, but this requires almost equal subsets of guessed bits; according to our experiments, this seems difficult to achieve. We also found some highly biased multibit differentials such as $([\Delta^4_1]_0 \oplus [\Delta^4_2]_9 \mid [\Delta^0_7]_{26})$ with bias $\varepsilon_d = -0.60$ for four rounds of Salsa20. However, exploiting multibit differentials, does not improve efficiency either. Note that an alternative approach to attack Salsa20/7 is to consider a 3-round biased differential, and observe it after going four rounds backward. This is however much more expensive than exploiting directly 4-round differentials. Unlike Salsa20, our exhaustive search showed no bias in 4-round ChaCha, be it with one, two, or three target output bits. This argues in favor of the faster diffusion of ChaCha. But surprisingly, when comparing the attacks on Salsa20/8 and ChaCha7, results suggest that after four rounds backward, key bits are more correlated with the target difference in ChaCha than in Salsa20. Nevertheless, ChaCha looks more trustful on the overall, since we could break up to seven ChaCha rounds against eight for Salsa20. For the variants with a 128-bit key, we can break up to seven Salsa20 rounds, and up to six ChaCha rounds.

4 Analysis of Rumba

This section describes our results for the compression function Rumba. Our goal is to efficiently find colliding pairs for R-round Rumba, i.e. input pairs (M, M') such that $\mathrm{RumbaR}(M) \oplus \mathrm{RumbaR}(M') = 0$. Note that, compared to our attacks on Salsa20 (where a single biased bit could be exploited in an attack), a collision attack targets all 512 bits (or a large subset of them for near-collisions).

4.1 Collisions and Preimages in Simplified Versions

We show here the weakness of two simplified versions of Rumba, respectively an iterated version with 2048-bit-input compression function, and the compression function without the final feedforward.

On the Role of Diagonal Constants. Rumba20 is fed with 1536 bits, copied in a 2048-bit state, whose remaining 512 bits are the diagonal constants. It is tempting to see these values as the IV of a derived iterated hash function, and use diagonal values as chaining variables. However, Bernstein implicitly warned against such a construction, when claiming that *"Rumba20 will take about twice as many cycles per eliminated byte as Salsa20 takes per encrypted byte"* [7]; indeed, the 1536-bit input should contain both the 512-bit chaining value and the 1024-bit message, and thus for a 1024-bit input the Salsa20 function is called four times (256 bits processed per call), whereas in Salsa20 it is called once for a 512-bit input. We confirm here that diagonal values should *not* be replaced by the chaining variables, by presenting a method for finding collisions within about $2^{128}/6$ trials, against 2^{256} with a birthday attack: Consider the following algorithm: pick an arbitrary 1536-bit message block M^0, then compute $\mathrm{Rumba}(M^0) = H_0 \| H_1 \| H_2 \| H_3$, and repeat this until two distinct 128-bit chunks H_i and H_j are equal—say H_0 and H_1, corresponding to the diagonal constants of F_0 and F_1 in the next round; hence, these functions will be identical in the next round. A collision can then be obtained by choosing two distinct message blocks $M^1 = M^1_0 \| M^1_1 \| M^1_2 \| M^1_3$ and $(M')^1 = M^1_1 \| M^1_0 \| M^1_2 \| M^1_3$, or $M^1 = M^1_0 \| M^1_0 \| M^1_2 \| M^1_3$ and $(M')^1 = (M'_0)^1 \| (M'_0)^1 \| M^1_2 \| M^1_3$. How fast is this method? By the birthday paradox, the amount of trials for finding a suitable M^0 is about $2^{128}/6$ (here 6 is the number of distinct sets $\{i, j\} \subset \{0, \ldots, 3\}$), while the construction of M^1 and $(M')^1$ is straightforward. Regarding the price-performance ratio, we do not have to store or sort a table, so the price is $2^{128}/6$—and this, for any potential filter function—while performance is much larger than one, because there are many collisions (one can choose 3 messages and 1 difference of 348 bits arbitrarily). This contrasts with the cost of 2^{256} for a serial attack on a 512-bit digest hash function.

On the Importance of Feedforward. In Davies-Meyer-based hash functions like MD5 or SHA-1, the final feedforward is an obvious requirement for one-wayness. In Rumba the feedforward is applied in each F_i, before an XOR of the four branches, and omitting this operation does not trivially lead to an inversion of the function, because of the incremental construction. However, as we will demonstrate, preimage resistance is not guaranteed with this setting. Let $F_i(M_i) = X_i^{20}$, $i = 0, \dots, 3$ and assume that we are given a 512-bit value H, and our goal is to find $M = (M_0, M_1, M_2, M_3)$ such that $\text{Rumba}(M) = H$. This can be achieved by choosing *random* blocks M_0, M_1, M_2, and set

$$Y = F_0(M_0) \oplus F_1(M_1) \oplus F_2(M_2) \oplus H \ . \tag{10}$$

We can find then the 512-bit state X_3^0 such that $Y = X_3^{20}$. If X_3^0 has the correct diagonal values (the 128-bit constant of F_3), we can extract M_3 from X_0^3 with respect to Rumba's definition. This randomized algorithm succeeds with probability 2^{-128}, since there are 128 constant bits in an initial state. Therefore, a preimage of an arbitrary digest can be found within about 2^{128} trials, against $2^{512/3}$ $(= 2^{512/(1+\log_2 4)})$ with the generalized birthday method.

4.2 Differential Attack

To obtain a collision for RumbaR, it is sufficient to find two messages M and M' such that

$$F_0(M_0) \oplus F_0(M_0') = F_2(M_2) \oplus F_2(M_2') \ , \tag{11}$$

with $M_0 \oplus M_0' = M_2 \oplus M_2'$, $M_1 = M_1'$ and $M_3 = M_3'$. The freedom in choosing M_1 and M_3 trivially allows to derive many other collisions (*multicollision*). We use the following notations for differentials: Let the initial states X_i and X_i' have the $\mathcal{ID}$ $\Delta_i^0 = X_i \oplus X_i'$ for $i = 0, \dots, 3$. After r rounds, the *observed* difference is denoted $\Delta_i^r = X_i^r \oplus (X_i')^r$, and the $\mathcal{OD}$ (without feedforward) becomes $\Delta_i^R = X_i^R \oplus (X_i')^R$. If feedforward is included in the $\mathcal{OD}$, we use the notation $\nabla_i^R = (X_i + X_i^R) \oplus (X_i' + (X_i')^R)$. With this notation, Eq. 11 becomes $\nabla_0^R = \nabla_2^R$, and if the feedforward operation is ignored in the F_i's, then Eq. 11 simplifies to $\Delta_0^R = \Delta_2^R$. To find messages satisfying Eq. 11, we use an R-round differential path of high-probability, with intermediate *target* difference δ^r after r rounds, $0 \le r \le R$. Note that the differential is applicable for both F_0 and F_2, thus we do not have to subscript the target difference. The probability that a random message pair with $\mathcal{ID}$ δ^0 conforms to δ^r is denoted p_r. To satisfy the equation $\Delta_0^R = \Delta_2^R$, it suffices to find message pairs such that the observed differentials equal the target one, that is, $\Delta_0^R = \delta^R$ and $\Delta_2^R = \delta^R$. The naive approach is to try about $1/p_r$ random messages each. This complexity can however be lowered down by:

- Finding constraints on the message pair so that it conforms to the difference δ^1 after one round with certainty (this will be achieved by *linearization*).
- Deriving message pairs conforming to δ^r from a single conforming pair (the message-modification technique used will be *neutral bits*).

Finally, to have $\nabla_0^R = \nabla_2^R$, we need to find message pairs such that $\nabla_0^R = \delta^R \oplus \delta^0$ and $\nabla_2^R = \delta^R \oplus \delta^0$ (i.e. the additions are not producing carry bits). Given a random message pair that conforms to δ^R, this holds with probability about 2^{-v-w} where v and w are the respective weights of the $\mathcal{ID}$ δ^0 and of the target $\mathcal{OD}$ δ^R

(excluding the linear MSB's). The three next paragraphs are respectively dedicated to finding an optimal differential, describing the linearization procedure, and describing the neutral bits technique.

Remark 3. One can observe that the constants of F_0 and F_2 are almost similar, as well as the constants of F_1 and F_3 (cf. Appendix A). To improve the generalized birthday attack suggested in [7], a strategy is to find a pair (M_0, M_2) such that $F_0(M_0) \oplus F_2(M_2)$ is biased in any c bits after R rounds (where $c \approx 114$, see [7]), along with a second pair (M_1, M_3) with $F_1(M_1) \oplus F_3(M_3)$ biased in the same c bits. The sum $F_0(M_0) \oplus F_2(M_2)$ can be seen as the feedforward $\mathcal{OD}$ of two states having an $\mathcal{ID}$ which is nonzero in some diagonal words. However, differences in the diagonal words result in a large diffusion, and this approach seems to be much less efficient than differential attacks for only one function F_i.

Finding a High-Probability Differential. We search for a *linear* differential over several rounds of Rumba, i.e. a differential holding with certainty when additions are replaced by XOR's, see [13]. The differential is independent of the diagonal constants, and it is expected to have high probability for genuine Rumba if the linear differential has low weight. An exhaustive search for suitable $\mathcal{ID}$'s is not traceable, so we choose another method: We focus on a *single column* in X_i, and consider the weight of the input (starting with the diagonal element, which must be zero). With a fixed relative position of the non-zero bits in this input, one can obtain an output of low weight after the first linear round of Rumba (i.e. using the linearized Eq. 3). Here is a list of the mappings (showing the weight only) which have at most weight 2 in each word of the input and output:

$$\begin{array}{llll}
g_1 : (0,0,0,0) \to (0,0,0,0) & \qquad & g_8 : & (0,1,2,0) \to (1,1,1,0) \\
g_2 : (0,0,1,0) \to (2,0,1,1) & & g_9 : & (0,1,2,2) \to (1,1,1,2) \\
g_3 : (0,0,1,1) \to (2,1,0,2) & & g_{10} : & (0,2,1,1) \to (0,1,0,0) \\
g_4 : (0,1,0,1) \to (1,0,0,1) & & g_{11} : & (0,2,1,2) \to (0,0,1,1) \\
g_5 : (0,1,1,0) \to (1,1,0,1) & & g_{12} : & (0,2,2,1) \to (0,1,1,1) \\
g_6 : (0,1,1,1) \to (1,0,1,0) & & g_{13} : & (0,2,2,1) \to (2,1,1,1) \\
g_7 : (0,0,2,1) \to (2,1,1,1) & & g_{14} : & (0,2,2,2) \to (2,0,2,0)
\end{array}$$

The relations above can be used to construct algorithmically a suitable $\mathcal{ID}$ with all 4 columns. Consider the following example, where the state after the first round is again a combination of useful rows: $(g_1, g_{10}, g_1, g_{11}) \to (g_1, g_2, g_4, g_1)$. After 2 rounds, the difference has weight 6 (with weight 3 in the diagonal words). There is a class of $\mathcal{ID}$'s with the same structure: $(g_1, g_{10}, g_1, g_{11})$, $(g_1, g_{11}, g_1, g_{10})$, $(g_{10}, g_1, g_{11}, g_1)$, $(g_{11}, g_1, g_{10}, g_1)$. The degree of freedom is large enough to construct these 2-round linear differentials: the positions of the nonzero bits in a single mapping g_i are symmetric with respect to rotation of words (and the required g_i have an additional degree of freedom). Any other linear differential constructed with g_i has larger weight after 2 rounds. Let $\Delta_{i,j}$ denote the difference of word $j = 0, \ldots, 15$ in state $i = 0, \ldots, 3$. For our attacks on Rumba, we will consider the following input difference (with optimal rotation, such that many MSB's are involved):

$$\begin{array}{llll}\Delta^0_{i,2} & = \mathtt{00000002} & \Delta^0_{i,8} & = \mathtt{80000000}\\ \Delta^0_{i,4} & = \mathtt{00080040} & \Delta^0_{i,12} & = \mathtt{80001000}\\ \Delta^0_{i,6} & = \mathtt{00000020} & \Delta^0_{i,14} & = \mathtt{01001000}\end{array}$$

and $\Delta^0_{i,j} = 0$ for all other words j. The weight of differences for the first four linearized rounds is as follows (the subscript of the arrows denotes the approximate probability p_r that a random message pair conforms to this differential for a randomly chosen value for diagonal constants):

$$\begin{pmatrix}0&0&1&0\\2&0&1&0\\1&0&0&0\\2&0&2&0\end{pmatrix} \xrightarrow[2^{-4}]{\mathsf{Round}} \begin{pmatrix}0&0&0&0\\0&0&0&0\\1&0&0&0\\1&0&1&0\end{pmatrix} \xrightarrow[2^{-7}]{\mathsf{Round}} \begin{pmatrix}0&0&0&0\\0&0&0&0\\1&1&2&0\\0&0&1&1\end{pmatrix} \xrightarrow[2^{-41}]{\mathsf{Round}} \begin{pmatrix}2&2&3&1\\0&3&4&2\\1&1&7&3\\1&1&1&6\end{pmatrix} \xrightarrow[2^{-194}]{\mathsf{Round}} \begin{pmatrix}8&3&2&4\\5&10&3&4\\9&11&13&7\\6&9&10&9\end{pmatrix}$$

With this fixed $\mathcal{ID}$, we can determine the probability that the $\mathcal{OD}$ obtained by genuine Rumba corresponds to the $\mathcal{OD}$ of linear Rumba. Note that integer addition is the only nonlinear operation. Each nonzero bit in the $\mathcal{ID}$ of an integer addition behaves linearly (i.e. it does not create or annihilate a sequence of carry bits) with probability 1/2, while a difference in the MSB is always linear. In the first round, there are only four bits with associated probability 1/2, hence $p_1 = 2^{-4}$ (see also the subsection on linearization). The other cumulative probabilities are $p_2 = 2^{-7}$, $p_3 = 2^{-41}$, $p_4 = 2^{-194}$. For 3 rounds, we have weights $v = 7$ and $w = 37$, thus the overall complexity to find a collision after 3 rounds is about $2^{41+37+7} = 2^{85}$. For 4 rounds, $v = 7$ and $w = 112$, leading to a complexity 2^{313}. The probability that feedforward behaves linearly can be increased by choosing low-weight inputs.

Linearization. The first round of our differential has a theoretical probability of $p_1 = 2^{-4}$ for a random message. This is roughly confirmed by our experiments, where exact probabilities depend on the diagonal constants (for example, we experimentally observed $p_1 = 2^{-6.6}$ for F_0, and $p_1 = 2^{-6.3}$ for F_2, the other two probabilities are even closer to 2^{-4}). We show here how to set constraints on the message so that the first round differential holds with certainty, using methods similar to the ones in [13].

Let us begin with the first column of F_0, where $c_{0,0} = x_{0,0} = \mathtt{73726966}$. In the first addition $x_{0,0} + x_{0,12}$, we have to address $\Delta^0_{0,12}$, which has a nonzero (and non-MSB) bit on position 12 (counting from 0). The bits of the constant are $[x_{0,0}]_{12-10} = (\mathtt{010})_2$, hence the choice $[x_{0,12}]_{11,10} = (\mathtt{00})_2$ is sufficient for linearization. This corresponds to $x_{0,12} \leftarrow x_{0,12} \wedge \mathtt{FFFF3FFF}$. The subsequent 3 additions of the first column are always linear as only MSB's are involved. Then, we linearize the third column of F_0, where $c_{0,2} = x_{0,10} = \mathtt{30326162}$. In the first addition $x_{0,10} + x_{0,6}$, we have to address $\Delta^0_{0,6}$, which has a nonzero bit on position 5. The relevant bits of the constant are $[x_{0,10}]_{5-1} = (\mathtt{10001})_2$, hence the choice $[x_{0,6}]_{4-1} = (\mathtt{1111})_2$ is sufficient for linearization. This corresponds to $x_{0,6} \leftarrow x_{0,6} \vee \mathtt{0000001E}$. In the second addition $z_{0,14} + x_{0,10}$, the updated difference $\Delta^1_{0,14}$ has a single bit on position 24. The relevant bits of the constant are $[x_{0,10}]_{24,23} = (\texttt{00})_2$, hence the choice $[z_{0,14}]_{23} = (\texttt{0})_2$ is sufficient. Notice that conditions on the updated words must be transformed to the initial state words. As $z_{0,14} = x_{0,14} \oplus (x_{0,10} + x_{0,6}) \lll 8$, we find the condition $[x_{0,14}]_{23} = [x_{0,10} + x_{0,6}]_{16}$. If we let both sides be zero, we have $[x_{0,14}]_{23} = (\texttt{0})_2$ or $x_{0,14} \leftarrow x_{0,14} \wedge \texttt{FF7FFFFF}$, and $[x_{0,10} + x_{0,6}]_{16} = (\texttt{0})_2$. As $[x_{0,10}]_{16,15} = (\texttt{00})_2$, we can choose $[x_{0,6}]_{16,15} = (\texttt{00})_2$ or $x_{0,6} \leftarrow x_{0,6} \wedge \texttt{FFFE7FFF}$. Finally, the third addition $z_{0,2} + z_{0,14}$ must be linearized with respect to the single bit in $\Delta^1_{0,14}$ on position 24. A sufficient condition for linearization is $[z_{0,2}]_{24,23} = (\texttt{00})_2$ and $[z_{0,14}]_{23} = (\texttt{0})_2$. The second condition is already satisfied, so we can focus on the first condition. The update is defined by $z_{0,2} = x_{0,2} \oplus (z_{0,14} + x_{0,10}) \lll 9$, so we set $[x_{0,2}]_{24,23} = (\texttt{00})_2$ or $x_{0,2} \leftarrow x_{0,2} \wedge \texttt{FE7FFFFF}$, and require $[z_{0,14} + x_{0,10}]_{15,14} = (\texttt{00})_2$. As $[x_{0,10}]_{15-13} = (\texttt{011})_2$, we can set $[z_{0,14}]_{15-13} = (\texttt{101})_2$. This is satisfied by choosing $[x_{0,14}]_{15-13} = (\texttt{000})_2$ or $x_{0,14} \leftarrow x_{0,14} \wedge \texttt{FFFF1FFF}$, and by choosing $[x_{0,10} + x_{0,6}]_{8-6} = (\texttt{101})_2$. As $[x_{0,10}]_{8-5} = (\texttt{1011})_2$, we set $[x_{0,6}]_{8-5} = (\texttt{1111})_2$ or $x_{0,6} \leftarrow x_{0,6} \vee \texttt{000001E0}$. Altogether, we fixed 18 (distinct) bits of the input, other linearizations are possible.

The first round of F_2 can be linearized with exactly the same conditions. This way, we save an average factor of 2^4 (additive complexities are ignored). This linearization with sufficient conditions does not work well for more than one round because of an avalanche effect of fixed bits. We lose many degrees of freedom, and contradictions are likely to occur.

Neutral Bits. Thanks to linearization, we can find a message pair conforming to δ^2 within about $1/(2^{-7+4}) = 2^3$ trials. Our goal now is to efficiently derive from such a pair many other pairs that are conforming to δ^2, so that a search for three rounds can start after the second round, by using the notion of neutral bits again (cf. §3.3). Neutral bits can be identified easily for a fixed pair of messages, but if several neutral bits are complemented in parallel, then the resulting message pair may not conform anymore. A heuristic approach was introduced in [9], using a *maximal 2-neutral set*. A 2-neutral set of bits is a subset of neutral bits, such that the message pair obtained by complementing any two bits of the subset in parallel also conform to the differential. The size of this set is denoted n. In general, finding a 2-neutral set is an NP-complete problem—the problem is equivalent to the Maximum Clique Problem from graph theory, but good heuristic algorithms for dense graphs exist, see e.g. [10]. In the case of Rumba, we compute the value n for different message pairs that conform to δ^2 and choose the pair with maximum n. We observe that about 1/2 of the 2^n message pairs (derived by flipping some of the n bits of the 2-neutral set) conform to the differential[2]. This probability p is significantly increased, if we complement at most $\ell \ll n$ bits of the 2-neutral set, which results in a message space (not contradicting with the linearization) of size about $p \cdot \binom{n}{\ell}$. At this point, a full collision for 3 rounds has a reduced theoretical complexity of $2^{85-7}/p = 2^{78}/p$

[2] In the case of SHA-0, about 1/8 of the 2^n message pairs (derived from the original message pair by complementing bits from the 2-neutral set) conform to the differential for the next round.

(of course, p should not be smaller than 2^{-3}). Since we will have $p > \frac{1}{2}$ for a suitable choice of ℓ, the complexity gets reduced from 2^{85} to less than 2^{79}.

4.3 Experimental Results

We choose a random message of low weight, apply the linearization for the first round and repeat this about 2^3 times until the message pairs conforms to δ^2. We compute then the 2-neutral set of this message pair. This protocol is repeated a few times to identify a message pair with large 2-neutral set:

- For F_0, we find the pair of states (X_0, X_0') of low weight, with 251 neutral bits and a 2-neutral set of size 147. If we flip a random subset of the 2-neutral bits, then the resulting message pair conforms to δ^2 with probability $\mathrm{Pr} = 0.52$.

$$X_0 = \begin{pmatrix} \texttt{73726966} & \texttt{00000400} & \texttt{00000080} & \texttt{00200001} \\ \texttt{00002000} & \texttt{6d755274} & \texttt{000001fe} & \texttt{02000008} \\ \texttt{00000040} & \texttt{00000042} & \texttt{30326162} & \texttt{10002800} \\ \texttt{00000080} & \texttt{00000000} & \texttt{01200000} & \texttt{636f6c62} \end{pmatrix}$$

- For F_2, we find the pair of states (X_2, X_2') of low weight, with 252 neutral bits and a 2-neutral set of size 146. If we flip a random subset of the 2-neutral bits, then the resulting message pair conforms to δ^2 with probability $\mathrm{Pr} = 0.41$.

$$X_2 = \begin{pmatrix} \texttt{72696874} & \texttt{00000000} & \texttt{00040040} & \texttt{00000400} \\ \texttt{00008004} & \texttt{6d755264} & \texttt{000001fe} & \texttt{06021184} \\ \texttt{00000000} & \texttt{00800040} & \texttt{30326162} & \texttt{00000000} \\ \texttt{00000300} & \texttt{00000400} & \texttt{04000000} & \texttt{636f6c62} \end{pmatrix}$$

Given these pairs for 2 rounds, we perform a search in the 2-neutral set by flipping at most 10 bits (that gives a message space of about 2^{50}), to find pairs that conform to δ^3. This step has a theoretical complexity of about 2^{34} for each pair (which was verified in practice). For example, in (X_0, X_0') we can flip the bits {59, 141, 150, 154, 269, 280, 294 ,425} in order to get a pair of states $(\bar{X}_0, \bar{X}_0')$ that conforms to δ^3.In the case of (X_2, X_2'), we can flip the bits {58, 63, 141, 271, 304, 317, 435, 417, 458, 460} in order to get a pair of states $(\bar{X}_2, \bar{X}_2')$ that conforms to δ^3.

$$\bar{X}_0 = \begin{pmatrix} \texttt{73726966} & \texttt{08000400} & \texttt{00000080} & \texttt{00200001} \\ \texttt{04400000} & \texttt{6d755274} & \texttt{000001fe} & \texttt{02000008} \\ \texttt{01002040} & \texttt{00000002} & \texttt{30326162} & \texttt{10002800} \\ \texttt{00000080} & \texttt{00000200} & \texttt{01200000} & \texttt{636f6c62} \end{pmatrix}$$

$$\bar{X}_2 = \begin{pmatrix} \texttt{72696874} & \texttt{84000000} & \texttt{00040040} & \texttt{00000400} \\ \texttt{0000a004} & \texttt{6d755264} & \texttt{000001fe} & \texttt{06021184} \\ \texttt{00008000} & \texttt{20810040} & \texttt{30326162} & \texttt{00000000} \\ \texttt{00000300} & \texttt{00080402} & \texttt{04001400} & \texttt{636f6c62} \end{pmatrix}$$

At this point, we have collisions for 3-round Rumba without feedforward, hence $\varDelta_0^3 \oplus \varDelta_2^3 = 0$. If we include feedforward for the above pairs of states, then $\nabla_0^3 \oplus \nabla_2^3$ has weight 16, which corresponds to a near-collision. Note that a near-collision indicates non-randomness of the reduced-round compression function (we assume a Gaussian distribution centered at 256). This near-collision of low weight was found by using a birthday-based method: we produce a list of pairs for F_0 that conform to δ^3 (using neutral bits as above), together with the corresponding value of ∇_0^3. The same is done for F_2. If each list has size N, then we can produce N^2 pairs of $\nabla_0^3 \oplus \nabla_2^3$ in order to identify near-collisions of low weight.

However, there are no neutral bits for the pairs $(\bar{X}_0, \bar{X}'_0)$ and $(\bar{X}_2, \bar{X}'_2)$ with respect to δ^3. This means that we cannot completely separate the task of finding full collisions with feedforward, from finding collisions without feedforward (and we can not use neutral bits to iteratively find pairs that conform to δ^4). To find a full collision after three rounds, we could perform a search in the 2-neutral set of (X_0, X'_0) and (X_2, X'_2) by flipping at most 20 bits. In this case, the resulting pairs conform to δ^2 with probability at least $\Pr = 0.68$, and the message space has a size of about 2^{80}. The overall complexity becomes $2^{78}/0.68 \approx 2^{79}$ (compared to 2^{85} without linearization and neutral bits). Then, we try to find near-collisions of low weight for 4 rounds, using the birthday method described above. Within less than one minute of computation, we found the pairs $(\bar{\bar{\bar{X}}}_0, \bar{\bar{\bar{X}}}'_0)$ and $(\bar{\bar{\bar{X}}}_2, \bar{\bar{\bar{X}}}'_2)$ such that $\nabla_0^4 \oplus \nabla_2^4$ has weight 129. Consequently, the non-randomness of the differential is propagating up to 4 rounds.

$$\bar{\bar{\bar{X}}}_0 = \begin{pmatrix} \texttt{73726966} & \texttt{00020400} & \texttt{00000080} & \texttt{00200001} \\ \texttt{00002400} & \texttt{6d755274} & \texttt{000001fe} & \texttt{02000008} \\ \texttt{00000040} & \texttt{00220042} & \texttt{30326162} & \texttt{10002800} \\ \texttt{00000080} & \texttt{00001004} & \texttt{01200000} & \texttt{636f6c62} \end{pmatrix}$$

$$\bar{\bar{\bar{X}}}_2 = \begin{pmatrix} \texttt{72696874} & \texttt{00001000} & \texttt{80040040} & \texttt{00000400} \\ \texttt{00008804} & \texttt{6d755264} & \texttt{000001fe} & \texttt{06021184} \\ \texttt{00000000} & \texttt{80800040} & \texttt{30326162} & \texttt{00000000} \\ \texttt{00000300} & \texttt{00000450} & \texttt{04000000} & \texttt{636f6c62} \end{pmatrix}$$

5 Conclusions

We presented a novel method for attacking reduced-round Salsa20 and ChaCha, inspired by correlation attacks and by the notion of neutral bits. This allows to give the first attack faster than exhaustive search on the stream cipher Salsa20/8 with a 256-bit key. For the compression function Rumba the methods of linearization and neutral bits are applied to a high probability differential to find collisions on 3-round Rumba within 2^{79} trials, and to efficiently find low weight near collisions on 3-round and 4-round Rumba.

Acknowledgments

The authors would like to thank Dan Bernstein for insightful comments on a preliminary draft, the reviewers of FSE 2008 who helped us to improve the clarity of the paper, and Florian Mendel for his proofreading. J.-Ph. Aumasson is supported by the Swiss National Science Foundation (SNF) under project number 113329. S. Fischer is supported by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center of the SNF under grant number 5005-67322. W. Meier is supported by Hasler Foundation (see http://www.haslerfoundation.ch) under project number 2005. C. Rechberger is supported by the Austrian Science Fund (FWF), project P19863, and by the European Commission through the IST Programme under Contract IST-2002-507932 ECRYPT.

References

1. Baignères, T., Junod, P., Vaudenay, S.: How far can we go beyond linear cryptanalysis? In: Lee, P.J. (ed.) ASIACRYPT 2004. LNCS, vol. 3329, pp. 432–450. Springer, Heidelberg (2004)
2. Bellare, M., Micciancio, D.: A new paradigm for collision-free hashing: Incrementality at reduced cost. In: Fumy, W. (ed.) EUROCRYPT 1997. LNCS, vol. 1233, pp. 163–192. Springer, Heidelberg (1997)
3. D.J. Bernstein. ChaCha, a variant of Salsa20, See also [8], http://cr.yp.to/chacha.html
4. Bernstein, D.J.: Salsa20 and ChaCha. eSTREAM discussion forum, May 11 (2007)
5. Bernstein, D.J.: Salsa20. Technical Report 2005/025, eSTREAM, ECRYPT Stream Cipher Project (2005), http://cr.yp.to/snuffle.html
6. Bernstein, D.J.: Salsa20/8 and Salsa20/12. Technical Report 2006/007, eSTREAM, ECRYPT Stream Cipher Project (2005)
7. Bernstein, D.J.: What output size resists collisions in a XOR of independent expansions? ECRYPT Workshop on Hash Functions (2007), http://cr.yp.to/rumba20.html
8. Bernstein, D.J.: ChaCha, a variant of Salsa20. In: SASC 2008 – The State of the Art of Stream Ciphers. ECRYPT (2008), http://cr.yp.to/rumba20.html
9. Biham, E., Chen, R.: Near-collisions of SHA-0. In: Franklin, M. (ed.) CRYPTO 2004. LNCS, vol. 3152, pp. 290–305. Springer, Heidelberg (2004)
10. Burer, S., Monteiro, R.D.C., Zhang, Y.: Maximum stable set formulations and heuristics based on continuous optimization. Mathematical Programming 64, 137–166 (2002)
11. Crowley, P.: Truncated differential cryptanalysis of five rounds of Salsa20. In: SASC 2006 – Stream Ciphers Revisited (2006)
12. ECRYPT. eSTREAM, the ECRYPT Stream Cipher Project, http://www.ecrypt.eu.org/stream
13. Fischer, S., Meier, W., Berbain, C., Biasse, J.-F., Robshaw, M.J.B.: Non-randomness in eSTREAM candidates Salsa20 and TSC-4. In: Barua, R., Lange, T. (eds.) INDOCRYPT 2006. LNCS, vol. 4329, pp. 2–16. Springer, Heidelberg (2006)
14. Junod, P., Vaudenay, S.: Optimal key ranking procedures in a statistical cryptanalysis. In: Johansson, T. (ed.) FSE 2003. LNCS, vol. 2887, pp. 235–246. Springer, Heidelberg (2003)

15. Siegenthaler, T.: Decrypting a class of stream ciphers using ciphertext only. IEEE Transactions on Computers 34(1), 81–85 (1985)
16. Tsunoo, Y., Saito, T., Kubo, H., Suzaki, T., Nakashima, H.: Differential cryptanalysis of Salsa20/8. In: SASC 2007 – The State of the Art of Stream Ciphers (2007)
17. Wagner, D.: A generalized birthday problem. In: Yung, M. (ed.) CRYPTO 2002. LNCS, vol. 2442, pp. 288–303. Springer, Heidelberg (2002)

A Constants

Here are the diagonal constants for Salsa20 and ChaCha (function Round) and for Rumba (functions F_0 to F_3).

	Round	F_0	F_1	F_2	F_3
c_0	61707865	73726966	6f636573	72696874	72756f66
c_1	3320646E	6d755274	7552646e	6d755264	75526874
c_2	79622D32	30326162	3261626d	30326162	3261626d
c_3	6B206574	636f6c62	6f6c6230	636f6c62	6f6c6230

Author Index