Lars R. Knudsen
Huapeng Wu (Eds.)

LNCS 7707

Selected Areas in Cryptography

19th International Conference, SAC 2012
Windsor, ON, Canada, August 2012
Revised Selected Papers

Springer

Lars R. Knudsen Huapeng Wu (Eds.)

Selected Areas in Cryptography

19th International Conference, SAC 2012
Windsor, ON, Canada, August 15-16, 2012
Revised Selected Papers

Springer

Volume Editors

Lars R. Knudsen
Technical University of Denmark
Dept. of Mathematics
Kgs. Lyngby, Denmark
E-mail: lars@ramkilde.com

Huapeng Wu
University of Windsor
Dept. of Electrical and Computer Engineering
Windsor, ON, Canada
E-mail: hwu@uwindsor.ca

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-35998-9 e-ISBN 978-3-642-35999-6
DOI 10.1007/978-3-642-35999-6
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012955349

CR Subject Classification (1998): E.3, D.4.6, K.6.5, F.2.1-2, C.2, H.4.3

LNCS Sublibrary: SL 4 – Security and Cryptology

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

Previously called the Workshop on Selected Areas in Cryptography, the Conference on Selected Areas in Cryptography (SAC) series was initiated in 1994, when the first workshop was held at Queen's University in Kingston. The SAC conference has been held annually since 1994 in various Canadian locations, including Calgary, Kingston, Montreal, Ottawa, Sackville, St. John's, Toronto, and Waterloo. More information on former SAC conferences can be found at the main SAC conferences site (http://sacconference.org/).

This volume contains the papers presented at SAC 2012, held on August 15–16, 2012 in Windsor, Canada. The objective of the conference is to present cutting edge research in the designated areas of cryptography and to facilitate future research through an informal and friendly conference setting.

The themes for the SAC 2012 conference were:

1. Design and analysis of symmetric key primitives and cryptosystems, including block and stream ciphers, hash functions, and MAC algorithms.
2. Efficient implementations of symmetric and public key algorithms.
3. Mathematical and algorithmic aspects of applied cryptology.
4. Light-weight authentication protocols.

There were 87 submissions. Each submission was reviewed by at least 3 program committee members. The committee decided to accept 24 papers and the acceptance rate was 24/87 = 27.6%. The program also included two invited talks.

We appreciate the hard work of the SAC 2012 Program Committee. We are also very grateful to the many others who participated in the review process: Mohamed Ahmed Abdelraheem, Jithra Adikari, Toru Akishita, Martin Albrecht, Hoda Alkhzaimi, Elena Andreeva, Kazumaro Aoki, Guido Bertoni, Zeeshan Bilal, Cline Blondeau, Céline Blondeau, Julia Borghoff, Yuri Borissov, Sebastien Canard, Murat Cenk, Qi Chai, Nadia El Mrabet, Junfeng Fan, Xinxin Fan, Sebastian Faust, Ewan Fleischmann, Thomas Fuhr, Louis Goubin, Robert Granger, Risto Hakala, Jens Hermans, Harunaga Hiwatari, Takanori Isobe, Pascal Junod, Abdel Alim Kamal, Kazuya Kamio, Dmitry Khovratovich, Aleksandar Kircanski, Mario Kirschbaum, Ilya Kizhvatov, Edward Knapp, Miroslav Knezevic, Noboru Kunihiro, Jooyoung Lee, Vadim Lyubashevsky, Kalikinkar Mandal, Alex May, Florian Mendel, Bart Mennink, Rafael Misoczki, Carlos Moreno, Mehran Mozaffari-Kermani, Nicolas Méloni, Tomislav Nad, Christophe Negre, Samuel Neves, Thomas Plos, Francesco Regazzoni, Alfredo Rial, Koichi Sakumoto, Stefaan Seys, Kyoji Shibutani, Dave Singelee, Hadi Soleimany, Paul Stankovski, Stefan Tillich, Elmar Tischhauser, Gilles Van Assche, Kerem Varici, Frederik Vercauteren, Lei Wang, Erich Wenger, Zhiqian Xu, Bo Zhu, and Martin Ågren. We apologize for any unintended errors or omissions in this list.

This conference was generously sponsored by the Department of Electrical and Computer Engineering, Faculty of Engineering, and the Office of Vice President - Research, University of Windsor.

Special thanks go to Dr. Carlisle Adams, University of Ottawa, Dr. Amr Youssef, Concordia University, and Dr. Ali Miri, Ryerson University for generously sharing their rich experiences in organizing SAC conferences.

We would like to thank Andria Ballo, Heather Quinn, Shelby Marchand, Yiruo He, and Shoaleh Hashemi Namin for their hard work in dealing with local arrangements. Our cordial thanks also go to EasyChair.org for providing support and convenience with regard to the paper-review and editing processes.

September 2012

Lars R. Knudsen
Huapeng Wu

Organization

Program Committee

Privacy Enhancing Technologies for the Internet (Invited Talk)

Ian Goldberg

Cheriton School of Computer Science
University of Waterloo
Waterloo, ON N2L 3G1 Canada
iang@cs.uwaterloo.ca

Abstract. The unprecedented communication power made available by the Internet has helped to spread freedom and democracy around the world. Unfortunately, there still exist regimes that restrict the flow of information, censor websites, and block some kinds of communication.

For more than a decade, privacy enhancing technologies have been used to allow people to communicate online while allowing them to control who can learn what they are saying and to whom they are speaking. Today, these same kinds of technologies are also being used to circumvent online censorship and allow global citizens to communicate freely.

In this talk, we will take a look at past, present, and upcoming privacy enhancing technologies for the Internet, and discuss their strengths and challenges.

Table of Contents

An All-In-One Approach to Differential Cryptanalysis for Small Block Ciphers[⋆]

Martin R. Albrecht[1] and Gregor Leander[2]

[1] INRIA, Paris-Rocquencourt Center, POLSYS Project
UPMC Univ Paris 06, UMR 7606, LIP6, F-75005, Paris, France
CNRS, UMR 7606, LIP6, F-75005, Paris, France
[2] DTU Mathematics, Department of Mathematics, Technical University of Denmark, 2800 Kgs. Lyngby, Denmark
malb@lip6.fr, G.Leander@mat.dtu.dk

Abstract. We present a framework that unifies several standard differential techniques. This unified view allows us to consider many, potentially all, output differences for a given input difference and to combine the information derived from them in an optimal way. We then propose a new attack that implicitly mounts several standard, truncated, impossible, improbable and possible future variants of differential attacks in parallel and hence allows to significantly improve upon known differential attacks using the same input difference. To demonstrate the viability of our techniques, we apply them to KATAN-32. In particular, our attack allows us to break 115 rounds of KATAN-32. For this, our attack exploits the non-uniformity of the difference distribution after 91 rounds which is 20 rounds more than the previously best known differential characteristic.

Keywords: symmetric cryptography, block cipher, differential attack.

1 Introduction

Designing a secure block cipher that, at the same time, is very efficient is still challenging. In particular, lightweight cryptography which recently received considerable attention from the cryptographic community calls for block ciphers that can be efficiently implemented even in very resource constrained devices. Designing secure ciphers for such tiny devices – e.g., RFID tags or sensor networks – requires, on the one hand, innovative design strategies and, on the other hand, perhaps compromises in the security level. One such constraint is the block size used in block ciphers. As the block size, along with the key size, greatly influences the required circuit size, block ciphers tailored to be implemented in small devices have a strong tendency to feature smaller block sizes compared to modern block ciphers mainly focusing on software such as the AES. While modern block ciphers focusing on software usually have a block size of no less than 128 bits, most ciphers designed for efficient implementations in hardware have block

⋆ This is an extended abstract of the work available as [1].

sizes of 64 bits or less (see for example PRESENT [8] or HIGHT [12]). A block cipher with a particular small block size of 32-bit is KATAN-32 [10] presented at CHES 2009.

Block ciphers with very small block sizes have some interesting characteristics. From the point of view of the attacker, when using the block cipher in counter mode, it is possible to distinguish the output from a random sequences faster. Similarly, an attacker can build a complete code book faster and time-memory tradeoffs are a greater concern. From the perspective of the designer, most statistical attacks like differential or linear cryptanalysis seem at first glance to become more difficult as the amount of data available to the attacker is much more restricted.

Finally, from a theoretical point of view, small block sizes provide the opportunity to understand well-established attacks better since computations involving the entire code-book are feasible. In particular, for differential cryptanalysis, it becomes feasible to compute the exact expected probabilities for many (sometimes all) differentials. This data then allows to study the behaviour of (classical) differential cryptanalysis and related techniques more precisely.

Yet, it is not obvious a priori how to provide an optimal unified view on these differentials even if this data is available. To provide an answer to this question, this work investigates the probability distribution of output differences under one (or many) input difference and provides an optimal way to use the non-uniform distribution of differences in an attack.

Prior Work: *Differential cryptanalysis* was first proposed by Biham and Shamir [4] and since became one of the most prominent tools in the analysis of block ciphers. Many improvements and extensions have been proposed in the past, we mention some of the most influential ones. Knudsen [15] and later Biham, Biryukov and Shamir [3] proposed to use differentials with zero probability, that is *impossible differential* attacks. Based on the work of Lai [17] *High-order differentials* were introduced in [16] and are most effective against ciphers where the algebraic degree can be limited. *Truncated differentials*, first mentioned in [16] can be seen as a collection of differentials and in some cases allow to push differential attacks one or two rounds further. *Boomerang attacks* can be viewed as special cases of second order differentials and are most efficient when the probability of any differential drops rapidly with an increasing number of rounds. Recently, *improbable differentials* have been suggested [22] as a natural extension of impossible differentials and have been successfully applied to the block cipher CLEFIA. Also recently, differential cryptanalysis was extended to *multi-differential cryptanalysis* in [6]. Finally, our application of the log-likelihood can be seen in the framework of [21].

Our Contribution: Abstractly, differential cryptanalysis exposes a non-uniform distribution of output differences given one (or several) input differences. This is also the point of view from which our investigation sets out. Phrased in these terms, recovering key information using differential techniques becomes the task of distinguishing between distributions, one for the right key and one for the wrong keys. However, usually the attacker does not have access to a full description of these distributions. In standard differential cryptanalysis only one output difference is considered and usually the probability of the best differential characteristic is considered in place of the probability of the output differential. Furthermore, for wrong keys it is assumed that the distribution is uniform.

In comparison the advantage of an attacker when dealing with small block-size ciphers become apparent. The attacker has, under mild assumptions, the ability to compute the parameters of those distributions precisely. Thus, the task is no longer to distinguish (essentially) unknown distributions, but distributions which are known completely. In particular, the usual hypotheses that wrong keys result in random permutations can be lifted. To this end, we first introduce a model to study and distinguish these distributions. As an important side effect, our framework unifies and generalises standard differential attacks, impossible differentials, improbable differentials and truncated differentials into one attack framework. Since our framework considers the distribution of all output differences it captures all techniques which exploit statistically significant subspaces of the output space.

We then propose a new attack based on this model that implicitly mounts several standard, truncated, impossible, improbable and possible future variants of differential attacks in parallel and hence allows to significantly improve upon known differential attacks using the same input difference. We stress that these "parallel applications" of various differential attacks are such that they are strictly better than those attacks considered independently. To demonstrate the viability of our model and attack, we apply our attack to two ciphers with small block sizes: the toy-cipher SmallPresent[4] and KATAN-32. For KATAN-32 we present the best known differential attack.[1] In particular, our attack allows us to break 115 rounds of KATAN-32, which is 37 rounds more than previous work [14], although we note that our attack requires considerably more resources than [14]. For this, our attack exploits the non-uniformity of the difference distribution after 91 rounds which is 20 rounds more than the previously best known differential characteristic. Since our results takes into account several standard techniques and still cover less than 1/2 of the cipher, they further strengthen our confidence in KATAN-32's resistance against differential attacks. For completeness, we also like to mention a recent preprint [13] using a meet-in-the-middle variant to recover the key for the full KATAN (slightly) faster than exhaustive search.

Furthermore, our model allows to combine many *input-* and output-differences which allows to reduce the data complexity compared to previous works significantly. This is mainly due to the fact that our approach almost naturally provides the optimal way of combining information from several input and output differences. This is the major difference between our work and [6].

We highlight that similar approaches have been independently developed by Blondeau, Gérard and Nyberg [7] and Murphy [20]. While these approaches also differ in some theoretical respects (such as using the likelihood instead of the

[1] Our attack is also the best known differential attack on SmallPresent[4].

likelihood ratio in the latter case), the main difference between these works and ours is that we put our model to practice and use it to improve upon known attacks.

2 Preliminaries and Notation

In this work we focus on block ciphers where the key is XORed to (parts of) the state. Let R_k denote one round function of a block cipher with (round)-key k, where without loss of generality the key is added in last. By R we denote the round function without the final key addition, that is $R_k(x) = R(x) \oplus k$. Moreover let $E_K : \mathbb{F}_2^n \to \mathbb{F}_2^n$ be the corresponding r round block cipher, where $K = (k_0, k_1, \dots, k_r)$ consist of all round keys. More precisely $E_K(x) = R_{k_r} \circ R_{k_{r-1}} \circ \cdots \circ R_{k_1}(x \oplus k_0)$ where k_0 is the whitening key. For a function $F : \mathbb{F}_2^n \to \mathbb{F}_2^n$ given an input difference δ and an output difference γ we denote

$$P_F(\delta, \gamma) := \mathbf{Pr}(F(X) \oplus F(X \oplus \delta) = \gamma)$$

for randomly uniformly chosen X. That is, $P_F(\delta, \gamma)$ is the probability of the differential $\delta \to \gamma$. Using N (unordered) pairs, the number of pairs following the given differential is denoted by $D_F^{(N)}(\delta, \gamma)$. The expected value of $D_F^{(N)}(\delta, \gamma)$ is $NP_F(\delta, \gamma)$ and we discuss below more precisely how $D_F^{(N)}(\delta, \gamma)$ is distributed.

Note that in the following N always denotes the number of (unordered) plaintext/ciphertext pairs used. As we use unordered pairs, using the full code book corresponds to choosing $N = 2^{n-1}$.

We consider the case where we assume E is a Markov cipher. A cipher E is a Markov cipher when the transitional probabilities for the output differences of round $r+1$ only depend on the output difference of round r. More precisely the round function has to satisfy [18]:

$$\mathbf{Pr}(R(X \oplus k) \oplus R(X \oplus \delta \oplus k) = \gamma \mid X = x_0) = P_R(\delta, \gamma)$$

for all choices of x_0 and uniformly random chosen subkeys k. If, furthermore, all round keys are independent, then one can compute the average value of $P_{E_K}(\delta, \gamma)$ over all possible keys by adding the probabilities for all differential characteristics included in the differential. This has first been formalised in [18] and is summarised in the next proposition.

Proposition 1. *For a function $E_K : \mathbb{F}_2^n \to \mathbb{F}_2^n = R_{k_r} \circ R_{k_{r-1}} \circ \cdots \circ R_{k_1}(x \oplus k_0)$ with input difference δ, output difference γ and $P_R(\gamma', \delta')$ the probability of the differential $\gamma' \to \delta'$ for the function R we have*

$$\begin{aligned}\tilde{P}_E(\delta, \gamma) &:= \frac{1}{\sharp K} \sum_K P_{E_K}(\delta, \gamma) \\ &= \sum_{\gamma_1, \dots, \gamma_{r-1}} P_R(\delta, \gamma_1) \left(\prod P_R(\gamma_i, \gamma_{i+1}) \right) P_R(\gamma_{r-1}, \gamma)\end{aligned}$$

The *hypothesis of stochastic equivalence* states (cf. [18]) that for almost all keys we expect $P_{E_K}(\delta,\gamma) \approx \tilde{P}_E(\delta,\gamma)$ which implies that $D_K^{(N)}(\delta,\gamma) \approx N\tilde{P}_E(\delta,\gamma)$ for almost all keys.

This approximation has to be understood as expected value taken over all expanded keys. However, for our purpose, we are not only interested in the expected value of the counter $D_{E_K}^{(N)}(\delta,\gamma)$ but moreover how these values are distributed. This was analysed in [11] and more recently in [5]. It turns out, considering $D_{E_K}^{(N)}(\delta,\gamma)$ as the results of N independent Bernoulli trials with success probability $\tilde{P}_E(\delta,\gamma)$ leads to a good model of the actual distribution. More precisely, denoting by $\mathcal{B}(n,p)$ the Binomial distribution with n tries and success probability p, the following is a reasonable approximation for the distribution of $D_{E_K}^{(N)}(\delta,\gamma)$.

Assumption 1 (cf. Theorem 14 in [11]). *The counter $D_{E_K}^{(N)}(\delta,\gamma)$ is distributed according to the Binomial distribution $\mathcal{B}(N,\tilde{P}_E(\delta,\gamma))$, that is*

$$\mathbf{Pr}(D_{E_K}^{(N)}(\delta,\gamma) = c) = \binom{N}{c}\tilde{P}_E(\delta,\gamma)^c(1-\tilde{P}_E(\delta,\gamma))^{N-c}$$

where the probability is taken over random keys K.

We note that we experimentally validated this assumption for all ciphers considered in this work although these ciphers are not Markov ciphers.

2.1 $\tilde{P}_E(\delta,\gamma)$ in Differential Cryptanalysis

In standard differential cryptanalysis the attacker attempts to find an input difference and an output difference such that $\tilde{P}_E(\delta,\gamma)$ is "sufficiently high", i.e., bounded away from uniform. In this case we can expect that, with high probability, for each key K there exist sufficiently many right pairs to mount an attack, i.e., to detect the bias of $\tilde{P}_E(\delta,\gamma)$. Traditionally, in a 1R attack on the cipher $R_{k_{r+1}} \circ E_K$ one (partially) decrypts the last round with all possible (partial) round keys and increases a counter for the current round key guess iff the computed difference fits the expected output difference γ of round r. Afterwards, the keys are ranked according to their counters, that is, the attacker first tries the key with the highest counter, than the one with the second highest counter, etc.

The success probability of a differential attack is usually computed under the *Wrong-Key Randomization Hypotheses*. The Wrong-Key Randomization Hypotheses (see for example [18]) states that for a wrong key guess the corresponding counter is distributed as for a random permutation. Using the notation established above the Wrong-Key Randomization Hypotheses can be stated as follows

Assumption 2 (Wrong-Key Randomization Hypotheses, cf. [18]). *For a wrong key guess the corresponding counter is distributed as for a random permutation, that is $D_{R_{k'}^{-1}\circ R_{k_{r+1}}\circ E_K}^{(N)}(\delta,\gamma) \sim \mathcal{B}(N,2^{-n})$ for all $k' \neq k_{r+1}$.*

2.2 Distinguishing Distributions

Following the above discussion on the distribution of counter values, it is natural to view a differential attack as a technique to find the value k_{r+1} which maximises the likelihood function corresponding to the right-key distribution (Maximum Likelihood Estimation). This estimation may take two distributions into account. For the right key guess, according to Assumption 1 the counter is distributed according to $\mathcal{B}(N, \tilde{P}_E(\delta,\gamma))$ while the counter of a wrong key guess is assumed (cf. Assumption 2) to be distributed accordingly to $\mathcal{B}(N, 2^{-n})$.

In this setting, the maximum likelihood estimation is equivalent to maximising the log-likelihood ratio of the two distributions under consideration. Indeed, by the Neyman-Pearson Lemma the log-likelihood ratio is the most poweful test to determine whether a sample comes from one of two distributions. Denoting $p = \tilde{P}_E(\delta,\gamma)$ and $q = 2^{-n}$, if a key K resulted in a counter value c one computes

$$l_k(c) := \log\left(\frac{\binom{N}{c}p^c(1-p)^{N-c}}{\binom{N}{c}q^c(1-q)^{N-c}}\right) = c\log\left(\left(\frac{p(1-q)}{q(1-p)}\right)\right) + N\cdot\log\left(\frac{1-p}{1-q}\right).$$

The key guesses are ranked according to their $l_k(c)$ values, that is, the key with highest $l_k(c)$ value is tested first. To simplify the computation one can equivalently rank the keys according to

$$l'_k(c) = c\cdot w \text{ where } w = \log\left(\frac{p(1-q)}{q(1-p)}\right),$$

as we are only interested in the relative value of $l_k(c)$.[2]

We may write $l_{k'}$ and $l'_{k'}$ for $l_{k'}(c)$ and $l'_{k'}(c)$ respectively if it is clear from the context which c we are referring to.

Now, observe that $l'_k(c)$ is monotone increasing iff $p > q$ (as in this case $w > 0$). Thus, if the expected counter for the right key is higher than for wrong key guesses then l'_k has the same ranking and the rankings accordingly to $l'_k(c)$ and c is the same. However, if $p < q$ the function is monotone decreasing (as $w < 0$) and the ranks get reversed. This corresponds to *improbable differentials* as defined in [22]. The special case where $p = 0$ corresponds to *impossible differentials* (as introduced in [15] and later used in [3]), as in this case for each counter $c \neq 0$ the value $l_k(c)$ is formally minus infinity. In the latter case we use the convention $w = -\infty$ and $0\cdot w = 0$. To conclude, we state the following observation.

Observation 1. *Ranking keys according to their maximum likelihood estimation as defined in Equation (1) unifies in a natural way standard differential attacks, impossible differentials and improbable differentials.*

As explained in the next section, it is this unified view that allows for a generalised attack that considers many (in principle all) counters $D^{(N)}_{E_K}(\delta,\gamma)$ simultaneously.

[2] As discussed below, this is actually equivalent to sorting according to the counters c in the case of $p > q$ and to $-c$ in the case of $p < q$.

3 The Attack Model

In this section, we present our attack in detail and provide formulas for computing the gain of our attack. In summary, we use many (or even all) counters $D_{E_K}^{(N)}(\delta,\gamma)$ for different δ and γ values simultaneously. We view those counters as samples from one out of two possible (this time multi-dimensional) distributions. One distribution corresponds to the correct round-key guess and the other to the wrong key guesses. Using many counters at the same time allows us to significantly improve the success probability (or – equivalently – reduce the data complexity) compared to standard differential attacks. Informally, and this is the major difference and biggest improvement over a related approach performed in [6], this allows us to perform several standard differential attacks and impossible (or more generally improbable) differential attacks at the same time. In our attacks these simultaneous differential attacks are weighted appropriately ensuring that we do not lose information compared to standard attacks. That is, considering more information never reduces the success probability but strictly improves it.

3.1 Multi-dimensional Distribution of $D_{E_K}^{(N)}(\delta,\gamma)$

While in general any subset of pairs of input/output differences could be considered, here we focus on the case where one input difference is fixed and we consider all possible output differences. In this case, we denote by

$$\mathcal{D}_{E_K}^{(N)}(\delta) = \left(D_{E_K}^{(N)}(\delta,1), D_{E_K}^{(N)}(\delta,2), \dots, D_{E_K}^{(N)}(\delta,2^n-1)\right)$$

the vector of all corresponding counters. As discussed in Section 2, each individual counter is distributed according to a binomial distribution $\mathcal{B}(N,\tilde{P}_E(\delta,\gamma))$. As each pair of the N pairs with input difference δ results in exactly one output difference, we have that

$$\sum_{\gamma} D_{E_K}^{(N)}(\delta,\gamma) = N.$$

Thus, assuming that this is the only dependency between the counter values, the vector $\mathcal{D}_{E_K}^{(N)}(\delta)$ follows a multinomial distribution with parameters N and $\tilde{P}_E(\delta) := \left(\tilde{P}_E(\delta,1),\dots,\tilde{P}_E(\delta,2^n-1)\right)$, denoted by $\mathcal{D}_{E_K}^{(N)}(\delta,\gamma) \sim \mathrm{Multi}(N,\tilde{P}_E(\delta))$.

Later in this work we present experimental evidence comparing the empirical and theoretical gain of the attack to justify this assumption for the ciphers considered in this work. We summarise our assumption on the behaviour below.

Assumption 3. *The vector of counters $\mathcal{D}_{E_K}^{(N)}(\delta)$ follows a multinomial distribution where each component is distributed according to Assumption 1 and*

$$\sum_{\gamma} D_{E_K}^{(N)}(\delta,\gamma) = N.$$

In contrast to previous works, we do not rely on the Wrong-Key Randomization Hypotheses (Assumption 2) for our attack. Before mounting our attack, the attacker has to compute the expected probability (or the expected counter value) for all possible output differences. If the attacker is able to do this, he is usually also able to compute the expected probability for wrong keys, that is compute the distribution of $D^{(N)}_{R^{-1}_{k'} \circ R_{k_{r+1}} \circ E_K}(\delta, \gamma)$ as this is essentially computing two more rounds. We note that even if k' differs from k in only a few bits, this affects at least one S-box and hence many output differences.

3.2 The Attack Algorithm

First, recall that the attack uses N plaintext/ciphertext pairs, to recover the secret key. Following the previous section, we assume that the attacker has – in an offline phase – computed the parameters of two distributions. Namely, vectors of parameters $p = (p_i)_i$ and $q = (q_i)_i$ such that

$$p_i = \tilde{P}_E(\delta, i) \tag{1}$$

$$q_i = \tilde{P}_{R^{-1} \circ R \circ E}(\delta, i). \tag{2}$$

That is, for a right key the vector of counters is a sample from the distribution $\mathrm{Dist}_1 = \mathrm{Multi}(N, p)$ and for the wrong keys sampled from the distribution $\mathrm{Dist}_2 = \mathrm{Multi}(N, q)$. After this pre-computation phase, the attack proceeds as follows . For all possible last round keys k', the attacker first computes the vector of difference counters $\mathcal{D}^{(N)}_{R^{-1}_{k'} \circ R_{k_{r+1}} \circ E_K}(\delta)$. That is, given the guess for the last round key, the attacker partially decrypts every ciphertext and for all output differences γ computes the number of pairs fulfilling the differential $\delta \to \gamma$. Next, the attacker estimates the likelihood that the vector was sampled from Dist_1. In our case, this is equivalent to computing the difference of the log-likelihood of the vector with respect to Dist_1 and with respect to Dist_2, i.e., to compute the log-likelihood-ratio.

Given that for a random variable X following a multinomial distribution $X \sim \mathrm{Multi}(M, p)$ it holds that

$$\mathbf{Pr}((X_1, \ldots, X_n) = (x_1, \ldots, x_n)) = \begin{cases} \frac{n!}{x_1! x_2! \ldots x_n!} p_1^{x_1} \cdots p_n^{x_n} & \text{if } \sum x_i = M \\ 0 & \text{else} \end{cases},$$

the log-likelihood-ratio is given by $l_{k'} = \sum_i D^{(N)}_{R^{-1}_{k'} \circ R_{k_{r+1}} \circ E_K}(\delta, i) \log\left(\frac{p_i}{q_i}\right)$ Thus, denoting $w_i = \log\left(\frac{p_i}{q_i}\right)$ one computes

$$l_{k'} = \sum_i w_i \cdot D^{(N)}_{R^{-1}_{k'} \circ R_{k_{r+1}} \circ E_K}(\delta, i).$$

This is a weighted extension of the case where one considers only one counter. As before these weights naturally capture various types of differential attacks, i.e., in each component one considers a standard differential, improbable or impossible differential attack. Furthermore, truncated differentials are captured in this model since these correspond to a sub-vector of $\mathcal{D}_{E_K}^{(N)}(\delta)$.[3]

The time complexity is $|K'| \cdot N$ where N is the number of pairs considered and $|K'|$ is the number of all last-round subkeys.

3.3 Computing the Gain of the Attack

What remains to be established is the efficiency of this attack. The key observation (cf. also [2]) is that the distribution of $l_{k'}$ can be well approximated by a normal distribution in the case where all values w_i are relatively close together. The case where all w_i are close to uniform is the most interesting case for our attack, as otherwise standard differential techniques, considering only one counter are sufficient to break the cipher. Recall that there are two distributions to be considered. First, there is a random variable (and a corresponding distribution) for the log-likelihood-ratio of the right key. We denote this random variable by $\mathcal{R}$ and it is defined as $\mathcal{R} = \sum_i w_i D_{E_K}^{(N)}(\delta, i)$. By Assumption 3 we expect $\mathcal{D}_{E_K}^{(N)}(\delta)$ to be multinomial distributed with parameters N and $(p_i)_i$, with p_i defined in Equation (1). Hence the expected value of $\mathcal{R}$ is given by $E(\mathcal{R}) = N \sum w_i p_i$. Using that the pairwise covariances for a multinomial distribution is known, the variance of $\mathcal{R}$ can be computed to be

$$\mathrm{Var}(\mathcal{R}) = N\left(\left(\sum_i w_i^2 p_i\right) - \left(\sum_i w_i p_i\right)^2\right).$$

Therefore, denoting by $\mathcal{N}(E, V)$ the normal distribution with expected value E and variance V, we will use the following approximation

$$\mathcal{R} \sim \mathcal{N}\left(N \sum w_i p_i, N\left(\left(\sum_i w_i^2 p_i\right) - \left(\sum_i w_i p_i\right)^2\right)\right)$$

which we will justify with experimental results later in this work.

For the wrong keys, we introduce a random variable $\mathcal{W}$ and, following the same lines of argumentation, we approximate the distribution of $\mathcal{W}$ with a normal distribution, as follows

$$\mathcal{W} \sim \mathcal{N}\left(N \sum w_i q_i, N\left(\left(\sum_i w_i^2 q_i\right) - \left(\sum_i w_i q_i\right)^2\right)\right)$$

with q_i as defined in Equation (2). This enables to estimate the gain of the attack. The gain is related to the probability that a wrong key candidate is

[3] We note, however, that in the case of truncated differential attacks we might have to assume that Assumption 2 holds.

ranked higher than the right key candidate. More precisely, if the task is to recover an n bit key and the rank of the correct key is r on average the gain is defined as $-\log_2 \frac{2r-1}{2^n}$. Given the probability that a wrong key is ranked higher than the right key the expected number of wrong keys ranked higher than the right key can be computed. This corresponds in turn to the expected rank of the right key.

For analyzing this, we assume that the right key value is sampled according to $\mathcal{R}$. As the normal distribution is symmetric, with a probability of $1/2$, the result is larger or equal to $E(\mathcal{R})$. For the wrong keys values are sampled from $\mathcal{W}$. For 50% percent of the keys, computing the gain is now reduced to computing the probability that $\mathcal{W} \geq E(\mathcal{R})$, as this corresponds to an upper bound on to the probability that a wrong key is ranked above the right key. Using the density function of $\mathcal{W}$, defined as

$$f_W(x) = \frac{1}{\sqrt{2\pi \operatorname{Var}(\mathcal{W})}} e^{-\frac{1}{2\operatorname{Var}(\mathcal{W})}(x-E(\mathcal{W}))^2}$$

this probability of a wrong key being ranked higher than the right key is given by $\mathbf{Pr}(\mathcal{W} \geq E(\mathcal{R})) = \int_{E(\mathcal{R})}^{\infty} f_W(x)$. Using the relation of the standard Normal distribution and the Gaussian error function, this can be rewritten as

$$\mathbf{Pr}(\mathcal{W} \geq E(\mathcal{R})) = \frac{1}{2}\left(1 - \operatorname{erf}\left(\frac{E(\mathcal{R}) - E(\mathcal{W})}{\sqrt{2\operatorname{Var}(\mathcal{W})}}\right)\right). \tag{3}$$

Concluding this part, we have now at hand an expression that allows us two compute the gain of the attack. Moreover, compared to computing the values of p_i and q_i, the time for evaluating the above expressions is negligible. We will make use of this in Section 4 where the model is applied to SmallPRESENT-[4] and KATAN. The experimental data given there justifies in turn the model for those two ciphers.

More Input Differences. A straight-forward extension which does not require any change to the analysis above is to use a different subset of input- and output-differences. In particular, the attack might benefit from not only using one vector $\mathcal{D}_{E_K}^{(N)}(\delta)$ but several such vectors for several choices of δ. We followed this approach in our experiments for SmallPRESENT-[4].

4 Application

In this section, we apply our framework to two blockciphers with very small block sizes. First, we consider SmallPRESENT-[4] to demonstrate the idea and then we consider reduced round variants of KATAN-32 for which we present the currently best known differential attack.

4.1 Toy Example SmallPRESENT-[4]

SmallPRESENT-[s] [19] is a small-scale (toy) cipher designed to aid the development and verification of cryptanalysis techniques. The cipher is an SP-network with s parallel 4-bit S-box applications. Hence the block size is $4s$. The permutation layer is a simple permutation of wires. We focus on SmallPRESENT-[4], the version with 16 bit block size, as this allows us to derive sufficient experimental data rather quickly. The S-box S is the same as for PRESENT (cf. [8]) itself and the round keys are independent. For more details we refer to [19]. A standard differential attack, with one round of partial decryption, seems feasible for not more than 7 rounds. By looking at all the whole vector of output differences, we are able to break 9 rounds with a significant gain. Moreover, compared to standard differential attacks, the data complexity for 7 rounds is reduced by a factor of 2^5. We summarise our findings for attacking $7, 8$ and 9 rounds in Table 1. All attacks in Table 1 are 1R attacks. Hence, the length of the differentials is 6, 7 and 8 respectively. In Table 1 we give the number of input differences considered, the values for $E(\mathcal{R}), V(\mathcal{R}), E(\mathcal{W}), V(\mathcal{W})$ and the number of right-key ranks smaller a than given threshold observed in 100 experiments (except for the last column, see below) compared with the number of such ranks predicted by our model (given in brackets in Table 1).

Table 1. Experimental Results for SmallPRESENT-[4]

#rounds	7	7	8	8	9	9
Data used	2^{16}	2^{9}	2^{16}	2^{16}	2^{16}	2^{16}
$\#\Delta$	1	1	1	5	1	60
$E(\mathcal{R})$	53.8210	0.8409	2.2250	9.7636	0.0570	1.5181
$V(\mathcal{R})$	124.0870	1.9388	4.6110	20.1537	0.1130	3.0441
$E(\mathcal{W})$	−47.2890	−0.7389	−2.1490	−9.4631	−0.0560	−1.5141
$V(\mathcal{W})$	84.2370	1.3162	4.1520	18.3502	0.1120	3.0203
#ranks $< 2^0$	100 (10000.00)	4 (1.10)	1 (2.70)	57 (61.95)	0 (6.81E-3)	1 (0.56)
#ranks $< 2^1$		4 (1.50)	2 (3.90)	65 (67.64)	0 (0.01)	2 (0.84)
#ranks $< 2^2$		4 (2.10)	3 (5.40)	73 (73.13)	0 (0.02)	2 (1.26)
#ranks $< 2^3$		4 (2.90)	5 (7.40)	79 (78.30)	0 (0.05)	2 (1.96)
#ranks $< 2^4$		4 (4.10)	8 (10.20)	82 (83.03)	0 (0.09)	2 (2.87)
#ranks $< 2^5$		4 (5.70)	11 (13.70)	84 (87.21)	1 (0.16)	5 (4.27)
#ranks $< 2^6$		5 (7.80)	16 (18.30)	88 (90.78)	1 (0.30)	8 (6.23)
#ranks $< 2^7$		9 (10.70)	25 (24.20)	91 (93.69)	1 (0.56)	14 (8.96)
#ranks $< 2^8$		14 (14.50)	31 (31.30)	95 (95.95)	3 (1.04)	22 (12.67)
#ranks $< 2^9$		19 (19.60)	42 (39.80)	96 (97.59)	3 (1.92)	28 (17.57)

For comparison, the best *6 round differential* for one active S-box is $\delta = \texttt{0x0007}, \gamma = \texttt{0x0404}$ where $\tilde{P}_E(\delta, \gamma) = 2^{-13.57}$ which is still sufficient to mount a standard differential attack. Consequently, our attack always succeeded as well. However, to go beyond standard differential attacks, even when using only

2^9 pairs, which for a standard differential attack would not be sufficient, we expect and observe a gain of more than 3.5 for 50% of the keys (cf. column 3 to Table 1). The best *7 round differential* for one active S-box is $\delta =$ `0x0007`, $\gamma =$ `0x0505` where $\tilde{P}_E(\delta, \gamma) = 2^{-15.39}$ which is not sufficient to mount a standard differential attack while our attack provides a gain of 5.97 for 50% of the keys. Using $N = 2^{14}$, $N = 2^{13}$, $N = 2^{12}$ and $N = 2^{11}$ we get a gain of 3.954, 2.821, 2.159 and 1.758 respectively. Using more than one input difference and the full code book, namely `0x0007`, `0x000f`, `0x0700`, `0x0070` and `0x0f00` we expect and observe (cf. column 5 of Table 1) a gain of 18.03 for 50% of the keys. Finally, the best *8 round differential* for one active S-box is $\delta =$ `0x0007`, $\gamma =$ `0x5055` where $\tilde{P}_E(\delta, \gamma) = 2^{-15.92}$.. Our attack has a gain of 1.44 for 50% of the keys(cf. column 6 of Table 1). Using all sixty input differences where one S-box is active in round one, we expect a gain of 4.625 which is better than exhaustive key search (over half the key space) by a factor of 3.625. Our experimental results for this case are presented in the last column of Table 1.

4.2 Application to KATAN-32

KATAN-32 is one member of a family of ciphers defined in [10]. It has a block-size of 32 bits, an 80 bit key and 254 rounds. The relatively small block-size of 32 bits makes it an interesting target for our technique. The plaintext is loaded into two registers of length 13 and 19, respectively. In each round, two bits of the registers are updated, involving one key bit each. We refer to [10] for more information. The currently best know differential attack on KATAN-32 is a conditional differential attack that can break up to 78 rounds given 2^{22} chosen plaintext/ciphertext pairs (see [14]). The best attack overall breaks the full cipher slightly faster than exhaustive key search [13]. Note that, for KTANTAN-32, which differs from KATAN-32 only in the key-scheduling, better attacks are known (see [9,23]) but they do not apply to KATAN-32.

Below, we always assume $\delta =$ `0x1006a880` which is the input difference for the best known differential characteristic which holds with probability 2^{-31} after 71 rounds disregarding any dependencies. Note, however, that the special structure of KATAN-32 means that in fact the first probabilistic difference only depends on the plaintext values and not on the key values.

We consider a $\ell = 24\text{R}$ attack below to maximise the number of rounds. This implies a computational cost of $2^{32} \cdot 2^{2\ell} = 2^{32+48} = 2^{80}$ partial decryptions in the online phase of the attack. Exhaustive search over half the key space would have to perform 2^{79} full encryptions where one full encryption costs roughly 4 partial decryptions. Hence, our attacks are twice as fast as exhaustive search. However, we emphasise that compared to exhaustive search the gain in our attack is significantly smaller.

71 + 24 Rounds of KATAN-32. The best output difference $\gamma =$ `0x00000008` has probability $\tilde{P}_E(\delta, \gamma) \approx 2^{-29.52}$, the output difference with the lowest probability is $\tilde{\gamma} =$ `0x00000080` with $\tilde{P}_E(\delta, \gamma) \approx 2^{-32.10}$. We get $E(\mathcal{R}) \approx 2505.211$, $V(\mathcal{R}) \approx$

5096.661, $E(\mathcal{W}) \approx -2467.448$, $V(\mathcal{W}) \approx 4868.280$. Which gives an expected gain of > 50 for 50% of the keys. We verified this estimate by considering the 16 differences with the highest probability. We compared randomly chosen right keys with randomly chosen wrong keys and always recovered the right key as the key with the highest rank.

91 + 24 Rounds of KATAN-32. The best output difference $\gamma =$ `0x00400000` has probability $\tilde{P}_E(\delta, \gamma) \approx 2^{-31.98}$, the output difference with the lowest probability is $\tilde{\gamma} =$ `0x02000000` with $\tilde{P}_E(\delta, \tilde{\gamma}) \approx 2^{-32.00}$. We get $E(\mathcal{R}) \approx 0.3695390$, $V(\mathcal{R}) \approx 0.7390803$, $E(\mathcal{W}) \approx -0.3695384$, $V(\mathcal{W}) \approx 0.7390745$. Which gives an expected gain of 2.3586180 for 50% of the keys. The expected gain for 50% of the keys for 92 and 94 rounds is 1.9220367 and 1.2306869 respectively.

5 Conclusions and Further Work

In this work we presented a unifying framework for several standard differential attacks. This unified view allows to naturally consider multiple differentials and by that improving upon known results. Our framework always provides better success probabilities than any of the combined differential attacks alone; although at the potential cost of increased computation time and memory. We demonstrated the viability of our approach by extending the the best differential for SmallPRESENT-[4] by two rounds and the best known differential for KATAN-32 by 20 rounds.

However, for many ciphers computing the distribution of counter values, i.e., $\mathcal{D}^{(N)}_{E_K}(\delta)$, is prohibitively expensive. For example, computing the Markov model exactly for KATAN-48 would require $\mathcal{O}(2^{48})$ memory which is well beyond what is feasible today. Yet, starting from one difference computing one or two rounds is usually feasible since only few output differences are possible after such a small number of rounds. It is thus possible to extend a standard differential attack using techniques discussed in this work. Instead of considering $\mathcal{D}^{(N)}_{E_K}(\delta)$ the attacker would consider $\mathcal{D}^{(N)}_{R_{k_r}}(\delta)$. Then, in the online phase of the attack counters are weighted accordingly to their distribution. We leave the details of such an approach open for further investigation.

Acknowledgements. We would like to thank Sean Murphy for helpful discussions on the log-likelihood ratio and its relation to the maximum likelihood estimation. We would also like to thank Céline Blondeau, Benoit Gérard and Kaisa Nyberg for helpful discussions on an earlier draft of this work. Furthermore, we would like to thank William Stein for allowing us to use his computers purchased under National Science Foundation Grant No. DMS-0821725. The second author gratefully acknowledges the support from the Danish National Research Foundation and the National Science Foundation of China(Grant No.11061130539) for the Danish-Chinese Center for Applications of Algebraic Geometry in Coding Theory and Cryptography. Finally, we would like to thank Michael Hortmann for introducing the authors to cryptography.

References

1. Albrecht, M.R., Leander, G.: An all-in-one approach to differential cryptanalysis for small block ciphers. Cryptology ePrint Archive, Report 2012/401 (2012), http://eprint.iacr.org/
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. 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)
4. Biham, E., Shamir, A.: Differential Cryptanalysis of DES-like Cryptosystems. In: Menezes, A., Vanstone, S.A. (eds.) CRYPTO 1990. LNCS, vol. 537, pp. 2–21. Springer, Heidelberg (1991)
5. Blondeau, C., Gérard, B.: Links between theoretical and effective differential probabilities: Experiments on PRESENT. In: Ecrypt II Workshop on Tools for Cryptanalysis (2010)
6. Blondeau, C., Gérard, B.: Multiple Differential Cryptanalysis: Theory and Practice. In: Joux, A. (ed.) FSE 2011. LNCS, vol. 6733, pp. 35–54. Springer, Heidelberg (2011)
7. Blondeau, C., Gérard, B., Nyberg, K.: Multiple Differential Cryptanalysis using LLR and χ^2 Statistics. Cryptology ePrint Archive, Report 2012/360 (2012), http://eprint.iacr.org/
8. Bogdanov, A.A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M., 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)
9. Bogdanov, A., Rechberger, C.: A 3-Subset Meet-in-the-Middle Attack: Cryptanalysis of the Lightweight Block Cipher KTANTAN. In: Biryukov, A., Gong, G., Stinson, D.R. (eds.) SAC 2010. LNCS, vol. 6544, pp. 229–240. Springer, Heidelberg (2011)
10. De Cannière, C., Dunkelman, O., Knežević, M.: KATAN and KTANTAN — A Family of Small and Efficient Hardware-Oriented Block Ciphers. In: Clavier, C., Gaj, K. (eds.) CHES 2009. LNCS, vol. 5747, pp. 272–288. Springer, Heidelberg (2009)
11. Daemen, J., Rijmen, V.: Probability distributions of correlation and differentials in block ciphers. Cryptology ePrint Archive, Report 2005/212 (2005), http://eprint.iacr.org/
12. Hong, D., Sung, J., Hong, S.H., Lim, J.-I., Lee, S.-J., Koo, B.-S., Lee, C.-H., Chang, D., Lee, J., Jeong, K., Kim, H., Kim, J.-S., Chee, S.: HIGHT: A New Block Cipher Suitable for Low-Resource Device. In: Goubin, L., Matsui, M. (eds.) CHES 2006. LNCS, vol. 4249, pp. 46–59. Springer, Heidelberg (2006)
13. Knellwolf, S.: Meet-in-the-Middle cryptanalysis of KATAN. In: ECRYPT Workshop on Lightweight Cryptography 2011 (to appear)
14. Knellwolf, S., Meier, W., Naya-Plasencia, M.: Conditional Differential Cryptanalysis of NLFSR-Based Cryptosystems. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 130–145. Springer, Heidelberg (2010)
15. Knudsen, L.: DEAL – a 128-bit block cipher. Technical report, Department of Informatics, University of Bergen, Norway (1998)

16. Lars, R.: Truncated and Higher Order Differentials. In: Preneel, B. (ed.) FSE 1994. LNCS, vol. 1008, pp. 196–211. Springer, Heidelberg (1995)
17. Lai, X.: Higher order derivatives and differential cryptanalysis. In: Communications and Cryptography (1994)
18. Lai, X., Massey, J.L.: Markov Ciphers and Differential Cryptanalysis. In: Davies, D.W. (ed.) EUROCRYPT 1991. LNCS, vol. 547, pp. 17–38. Springer, Heidelberg (1991)
19. Leander, G.: Small scale variants of the block cipher PRESENT. Cryptology ePrint Archive, Report 2010/143 (2010), `http://eprint.iacr.org/`
20. Murphy, S.: The analysis of simultaneous differences in Differential Cryptanalysis (2011), `http://www.isg.rhul.ac.uk/~sean/SimDiffA.pdf`
21. Murphy, S., Piper, F., Walker, M., Wild, P.: Likelihood estimation for block cipher keys (1995), `http://www.isg.rhul.ac.uk/~sean/maxlik.pdf`
22. Tezcan, C.: The Improbable Differential Attack: Cryptanalysis of Reduced Round CLEFIA. In: Gong, G., Gupta, K.C. (eds.) INDOCRYPT 2010. LNCS, vol. 6498, pp. 197–209. Springer, Heidelberg (2010)
23. Wei, L., Rechberger, C., Guo, J., Wu, H., Wang, H., Ling, S.: Improved Meet-in-the-Middle cryptanalysis of KTANTAN. Cryptology ePrint Archive, Report 2011/201 (2011), `http://eprint.iacr.org/`

A New Method for Solving Polynomial Systems with Noise over $\mathbb{F}_2$ and Its Applications in Cold Boot Key Recovery[⋆]

Zhenyu Huang and Dongdai Lin

State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
{huangzhenyu,ddlin}@iie.ac.cn

Abstract. The family of Max-PoSSo problems is about solving polynomial systems with noise, and is analogous to the well-known Max-SAT family of problems when the ground field is $\mathbb{F}_2$. In this paper, we present a new method called **ISBS** for solving the family of Max-PoSSo problems over $\mathbb{F}_2$. This method is based on the ideas of incrementally solving polynomial system and searching the values of polynomials with backtracking. The **ISBS** method can be combined with different algebraic methods for solving polynomial systems, such as the Gröbner Basis method or the Characteristic Set(CS) method. By combining with the CS method, we implement **ISBS** and apply it in Cold Boot attacks. A Cold Boot attack is a type of side channel attack in which an attacker recover cryptographic key material from DRAM relies on the data remanence property of DRAM. Cold Boot key recovery problems of block ciphers can be modeled as Max-PoSSo problems over $\mathbb{F}_2$. We apply the **ISBS** method to solve the Cold Boot key recovery problems of AES and Serpent, and obtain some experimental results which are better than the existing ones.

Keywords: polynomial system with noise, Max-PoSSo, Cold Boot attack, boolean equations, Characteristic Set method, AES, Serpent.

1 Introduction

Solving polynomial system with noise, which means finding an optimal solution from a group of polynomials with noise, is a fundamental problem in several areas of cryptography, such as algebraic attacks, side-channel attacks and the cryptanalysis of LPN/LWE-based schemes. In computation complexity field, this problem is also significant and called the maximum equation satisfying problem [7,14]. In the general case, this problem is NP-hard even when the polynomials are linear. In [1], the authors classified this kind of problems into three categories:

[⋆] This work was in part supported by National 973 Program of China under Grants No. 2013CB834203 and No. 2011CB302400, the National Natural Science Foundation of China under Grant No. 60970152 and the "Strategic Priority Research Program" of the Chinese Academy of Sciences under Grant No. XDA06010701.

Max-PoSSo, Partial Max-PoSSo, and Partial Weighted Max-PoSSo, and called them the family of Max-PoSSo problems. Moreover, they presented a model by which they can convert the Cold Boot key recovery problems of block ciphers into the family of Max-PoSSo problems. The Cold Boot key recovery problems originated from a side channel attack which is called the Cold Boot attack[8]. In a Cold Boot attack, an attacker with physical access to a computer is able to retrieve sensitive information from a running operating system after using a cold reboot to restart the machine from a completely “off” state. The attack relies on the data remanence property of DRAM to retrieve memory contents which remain readable in the seconds to minutes after power has been removed. Furthermore, the time of retention can be potentially increased by reducing the temperature of memory. Thus, data in memory can be used to recover potentially sensitive information, such as cryptographic keys. Due to the nature of the Cold Boot attack, it is realistic to assume that only decayed image of the data in memory can be available to the attacker, which means a fraction of memory bits will be flipped. Therefore, the most important step of the Cold Boot attack is recovering the original sensitive information from the decayed data.

In the case of block cipher, the sensitive information is the original key, and the decayed data is likely to be the round keys, which are generated from the origin key by the key schedule operation. Thus the Cold Boot key recovery problem of block cipher is recovering the origin key from the decayed round keys. Intuitively, every bit of round keys corresponds to a boolean polynomial equation with the bits of origin key as its variables. Then all bits of these round keys correspond to a boolean polynomial system. However, because of the data decay this polynomial system has some noise. In general case, these polynomials can be seen as random ones, so a random assignment may satisfy about half of them. If the percentage of the decayed bits is smaller than 50%, an assignment satisfying the maximum number of these polynomials may be equal to the origin key with high probability. By this way, we can model the Cold Boot key recovery problem of block cipher as the Max-PoSSo problem over $\mathbb{F}_2$, which is finding the optimal solution of a polynomial system with noise.

As mentioned before, the general Max-PoSSo problem over $\mathbb{F}_2$ is NP-hard. A natural way of solving Max-PoSSo problems over $\mathbb{F}_2$ is converting them into their SAT equivalents and then solve them by Max-SAT solvers. However, this method has a disadvantages that the original algebraic structure is destroyed. In [1], the authors converted the Max-PoSSo problems over $\mathbb{F}_2$ into mixed integer programming problems, and used the **MIP** solver **SCIP** to solve them. They presented some experimental results about attacking AES and Serpent. Their attack result about Serpent is a new result, and they showed that comparing with generic combinatorial approach their attack is much better.

The main contribution of this paper is that we propose a new method called **ISBS** for solving the family of Max-PoSSo problems over $\mathbb{F}_2$. The basic idea of **ISBS** is searching the values of polynomials. Precisely speaking, given a polynomials system with noise $\{f_1, f_2, \ldots, f_m\}$, we try to solve polynomial systems $\{f_1 + e_2, f_2 + e_2, \ldots, f_m + e_m\}$, where $\{e_1, e_2, \ldots, e_m\}$

can be equal to $\{0, 0, \ldots, 0\}, \{1, 0, \ldots, 0\}, \ldots, \{1, 1, \ldots, 1\}$. The solution of a $\{f_1+e_2, f_2+e_2, \ldots, f_m+e_m\}$ with $\{e_1, e_2, \ldots, e_m\}$ having the smallest Hamming-weight is the solution of the Max-PoSSo problem.

In the **ISBS** method, we combine the above idea with the ideas of incrementally solving $\{f_1 + e_1, f_2 + e_2, \ldots, f_m + e_m\}$ and searching $\{e_1, e_2, \ldots, e_m\}$ with backtracking. By this way, we can cut off a lot of branches when searching the values of $\{e_1, e_2, \ldots, e_m\}$. Furthermore, with the incremental solving method, we can use the former results to derive the latter ones, by which we can reduce a lot of computation.

In order to further improve the efficiency of **ISBS**, we combine it with an algebraic method for solving polynomial system which is called the Characteristic Set(CS) Method. In the field of symbolic computation, the CS method is an important tool for studying polynomial, algebraic differential, and algebraic difference equation systems. Its idea is reducing equation systems in general form to equation systems in the form of triangular sets. This method was introduced by Ritt and the recent study of it was inspired by Wu's seminal work on automated geometry theorem proving [13]. In [2], the CS method was firstly extended to solve polynomial equations in boolean ring. In [6], it was further extended to solve polynomial equations in general finite fields, and an efficient variant of the CS method called the **MFCS** algorithm was proposed and systematically analyzed. **MFCS** is an algorithm for solving boolean polynomial system, and it already had some applications in cryptanalysis[9]. **MFCS** has some advantage in incrementally solving polynomial system, thus we implemented **ISBS** with the **MFCS** algorithm.

Furthermore, we used **ISBS** to solve some Cold Boot key recovery problems of AES and Serpent, and compared our experimental results with those in [1]. From these results, we showed that by solving these problems with **ISBS** the success rate of recovering the origin key is higher and the average running time is shorter.

The rest of this paper is organized as follows. In Section 2, we introduce the family of Max-PoSSo problems and its relation with Cold Boot key recovery. In Section 3, we present the **ISBS** method and simply introduce the **MFCS** algorithm. In Section 4, our experimental results of attacking AES and Serpent are shown. In Section 5, the conclusions are presented. In Appendix, we present some tricks we used in solving the symmetric noise problems.

2 The Family of Max-PoSSo Problems and the Cold Boot Problem

In this section we will introduce the family of Max-PoSSo problems and its relationship with the Cold Boot key recovery problem.

2.1 The Family of Max-PoSSo Problems

Let $\mathbb{F}$ be a field, and $\mathbb{P} = \{f_1, \ldots, f_m\} \subset \mathbb{F}[x_1, \ldots, x_n]$ is a polynomial system. The *polynomial system solving (PoSSo)* problem is finding a solution $(x_1, \ldots, x_n) \in \mathbb{F}^n$ such that $\forall f_i \in \mathbb{P}$, we have $f_i(x_1, \ldots, x_n) = 0$.

By *Max-PoSSo*, we mean the problem of finding any $(x_1, \ldots, x_n) \in \mathbb{F}^n$ that satisfies the maximum number of polynomials in $\mathbb{P}$. The name "Max-PoSSo" was first proposed in [1]. In the computational complexity field, this problem is sometimes called the maximum equation satisfying problem [7,14]. Obviously, Max-PoSSo is at least as hard as PoSSo. Moreover, whether the polynomials in $\mathbb{P}$ are linear or not, Max-PoSSo is a NP-hard problem.

Besides Max-PoSSo, in [1], the authors introduced another two similar problems: Partial Max-PoSSo and Partial Weighted Max-PoSSo.

Partial Max-PoSSo problem is finding a point $(x_1, \ldots, x_n) \in \mathbb{F}^n$ such that for two sets of polynomials $\mathcal{H}, \mathcal{S} \in \mathbb{F}[x_1, \ldots, x_n]$, we have $f(x_1, \ldots, x_n) = 0$ for all $f \in \mathcal{H}$, and the number of polynomials $f \in \mathcal{S}$ with $f(x_1, \ldots, x_n) = 0$ is maximised. It is easy to see that Max-PoSSo is a special case of Partial Max-PoSSo when $\mathcal{H} = \emptyset$.

Let $\mathcal{H}, \mathcal{S}$ are two polynomial sets in $\mathbb{F}[x_1, \ldots, x_n]$. $\mathcal{C} : \mathcal{S} \times \mathbb{F}^n \to \mathbb{R}_{\geq 0}, (f, x) \to v$ is a cost function. $v = 0$ if $f(x) = 0$ and $v > 0$ if $f(x) \neq 0$. *Partial Weighted Max-PoSSo* denotes the problem of finding a point $(x_1, \ldots, x_n) \in \mathbb{F}^n$ such that $\forall f \in \mathcal{H}$ we have $f(x) = 0$ and $\sum_{f \in \mathcal{S}} \mathcal{C}(f, x)$ is minimised. Obviously, Partial Max-PoSSo is Partial Weighted Max-PoSSo when $\mathcal{C}(f, x) = 1$ if $f(x) \neq 0$ for all f.

In the definition of the above four problems, the ground field $\mathbb{F}$ can be any field. However, in the following of this paper, we focus on the case of $\mathbb{F} = \mathbb{F}_2$, where $\mathbb{F}_2$ is the finite field with elements 0 and 1, and this is the most common case in cryptanalysis.

2.2 Cold Boot Key Recovery as Max-PoSSo

The Cold Boot key recovery problem was first proposed and discussed in the seminal work of [8]. In [1], the authors built a new mathematical model for Cold Boot key recovery problem of block cipher, by which they can convert the Cold Boot problem into the Partial Weighted Max-PoSSo problem. In the following, we extract the definition of this model from [1].

First, according to [8], we should know that bit decay in DRAM is usually asymmetric: bit flips $0 \to 1$ and $1 \to 0$ occur with different probabilities, depending on the "ground state". The Cold Boot problem of block cipher can be defined as follows.

Consider an efficiently computable vectorial Boolean function $\mathcal{KS} : \mathbb{F}_2^n \to \mathbb{F}_2^N$ where $N > n$, and two real numbers $0 \leq \delta_0, \delta_1 \leq 1$. Let $K = \mathcal{KS}(k)$ be the image for some $k \in \mathbb{F}_2^n$, and K_i be the i-th bit of K. Given K, we compute $K' = (K'_0, K'_1, \ldots, K'_{N-1}) \in \mathbb{F}_2^N$ according to the following probability distribution:

$$Pr[K'_i = 0 | K_i = 0] = 1 - \delta_1, Pr[K'_i = 1 | K_i = 0] = \delta_1,$$

$$Pr[K'_i = 1 | K_i = 1] = 1 - \delta_0, Pr[K'_i = 0 | K_i = 1] = \delta_0.$$

Then we can consider such a K' as a noisy output of $\mathcal{KS}$ for some $k \in \mathbb{F}_2^n$, with the probability of a bit 1 in K flipping to 0 is δ_0 and the probability of a bit 0 in K flipping to 1 is δ_1. It follows that a bit $K'_i = 0$ of K' is correct with probability

$$Pr[K_i = 0|K_i = 0] = \frac{Pr[K'_i = 0|K_i = 0]Pr[K_i = 0]}{Pr[K'_i = 0]} = \frac{(1-\delta_1)}{(1-\delta_1+\delta_0)}.$$

Likewise, a bit $K'_i = 1$ of K' is correct with probability $\frac{(1-\delta_0)}{(1-\delta_0+\delta_1)}$. We denote these values by Δ_0 and Δ_1 respectively. Now assume we are given a description of the function $\mathcal{KS}$ and a vector $K' \in \mathbb{F}_2^N$ obtained by the process described above. Furthermore, we are also given a control function $E : \mathbb{F}_2^n \rightarrow \{True, False\}$ which returns $True$ or $False$ for a candidate k. The task is to recover k such that $\mathcal{E}(k)$ returns $True$. For example, $\mathcal{E}$ could use the encryption of some known data to check whether k is the original key. In the context of this work, we can consider the function $\mathcal{KS}$ as the key schedule operation of a block cipher with n-bit keys. The vector K is the result of the key schedule expansion for a key k, and the noisy vector K' is obtained from K due to the process of memory bit decay.

We can consider the Cold Boot Problem as a Partial Weighted Max-PoSSo problem over $\mathbb{F}_2$. Let $F_{\mathcal{K}}$ be an equation system corresponding to $\mathcal{KS}$ such that the only pairs (k, K) that satisfy $F_{\mathcal{K}}$ are any $k \in \mathbb{F}_2^n$ and $K = \mathcal{KS}(k)$. In our task however, we need to consider $F_{\mathcal{K}}$ with k and K'. Assume that for each noisy output bit K'_i there is some $f_i \in F_{\mathcal{K}}$ of the form $g_i + K'_i$ where g_i is some polynomial. Furthermore assume that these are the only polynomials involving the output bits ($F_{\mathcal{K}}$ can always be brought into this form) and denote the set of these polynomials by $\mathcal{S}$. Denote the set of all remaining polynomials in $F_{\mathcal{K}}$ as $\mathcal{H}$, and define the cost function $\mathcal{C}$ as a function which returns

$$\frac{1}{1-\Delta_0},\ \text{for}\ K'_i = 0, f(x) \neq 0; \quad \frac{1}{1-\Delta_1},\ \text{for}\ K'_i = 1, f(x) \neq 0; \quad 0,\ \text{otherwise.}$$

This cost function returns a cost proportional to the inverse of the probability that a given value is correct. Finally, let $F_{\mathcal{E}}$ be an equation system that is only satisfiable for $k \in \mathbb{F}_2^n$ for which $\mathcal{E}$ returns True. This will usually be an equation system for one or more encryptions. Add the polynomials in $F_{\mathcal{E}}$ to $\mathcal{H}$.[2] Then $\mathcal{H}, S, C$ define a Partial Weighted Max-PoSSo problem. Any optimal solution x to this problem is a candidate solution for the Cold Boot problem.

3 A New Method for Solving Max-PoSSo Problems

3.1 The Incremental Solving and Backtracking Search (ISBS) Method

In this part, we will introduce the **ISBS** method for solving the family of Max-PoSSo problems. First, let's show our idea. Almost all existing algorithms for solving Max-PoSSo problems are based on the idea of searching the values of

[2] Actually, in our following attacks on AES and Serpent, we didn't add $F_{\mathcal{E}}$ into $\mathcal{H}$. The reason is in our method we need to solve $\mathcal{H}$ first, and if $F_{\mathcal{E}}$ is added solving $\mathcal{H}$ will be extremely inefficient.

variables, such as the Max-SAT solvers. Our idea is based on another direction which is searching the values of polynomials. Now let's show it specifically.

Given a noisy polynomial set $\mathbb{P} = \{f_1, f_2, \ldots, f_m\}$. For every vector $E = [e_1, e_2, \ldots, e_m] \in \mathbb{F}_2^n$, we can solve the polynomial system $\{f_1+e_1, f_2+e_2, \ldots, f_n+e_m\}$ by an algebraic method. We can exhaustively searching E in the order of increasing Hamming weight and solve the corresponding polynomial system for each E. If the corresponding equation set of some E have a solution, then it is the solution of the Max-PoSSo problem.

To improve the solving and searching efficiencies, we combine the incremental solving method and backtracking search method with the above idea. For a polynomial set $\mathbb{P}$, we can solve it by some algebraic method, such as the Characteristic Set(CS) method [2,6] or the Gröbner Basis method [4,5]. We denote the output results of such a solving algorithm with input $\mathbb{P}$ by Result($\mathbb{P}$). From Result($\mathbb{P}$), we can deriver all the solutions of $\mathbb{P}$ easily. We remind the reader that, for different methods, Result($\mathbb{P}$) can be different. For example, if we use the CS method to solve $\mathbb{P}$, Result($\mathbb{P}$) $= \cup_i \mathcal{A}_i$ is a group of triangular sets(A triangular set $\mathcal{A}_i$ is a polynomial set which can be easily solved, and its precise definition will be given in next section). If we use the Gröbner Basis method to solve $\mathbb{P}$, Result($\mathbb{P}$) is the Gröbner Basis of idea $< \mathbb{P} >$. When this polynomial system has no solution, we set Result($\mathbb{P}$) $= \{1\}$. Obviously, given Result($\mathbb{P}$) and a polynomial g, we can achieve Result($\{$Result($\mathbb{P}$)$, g\}$). For example, for the CS method, we need to compute each Result($\mathcal{A}_i, g$) and return the union of them. For the Gröbner Basis method, we need to compute the Gröbner Basis of idea generated by the new polynomial set. Therefore, given a polynomial system $\mathbb{P} = \{f_1, f_2, \ldots, f_m\}$, we can get Result($\mathbb{P}$) by computing Result($\{f_1\}$), Result($\{$Result($\{f_1\}$)$, f_2\}$), ... This is the incremental solving method.

In the **ISBS** method, first we try to incrementally solve $\{f_1+e_1, f_2+e_2, \ldots, f_i+e_i\}$ for i from 1 to m with each $e_i = 0$. If Result($\{f_1+e_1, f_2+e_2, \ldots, f_i+e_i\}$) $= 1$ for some e_i, we flip e_i to 1 and solve this new $\{f_1 + e_1, f_2 + e_2, \ldots, f_i + e_i\}$. At last, we will obtain a candidate Result($\{f_1 + e_1, f_2 + e_2, \ldots, f_m + e_m\}$) where $[e_1, \ldots, e_m] = [e'_1, \ldots, e'_m]$. Then, in order to obtain a better candidate, we search all the possible values of $[e_1, \ldots, e_m]$ with backtracking based on the value $[e'_1, \ldots, e'_m]$. That is we flip the value of e'_i for i from m to 1, and try to incrementally solve all the new systems $\{f_1+e'_1, \ldots, f_i+e'_i+1, f_{i+1}+e_{i+1}, \ldots, f_m+e_m\}$. Finally, we will find the optimal solution after searching all the possible $[e_1, \ldots, e_m]$.

In the following Algorithm 1, we will present the **ISBS** method specifically.

Here we should introduce a notation. For a polynomial set $\mathbb{P} = \{f_1, \ldots, f_m\}$, We use Zero($\mathbb{P}$) to denote the common zeros of the polynomials in $\mathbb{P}$ in the affine space $\mathbb{F}_2^n$, that is,

$$\text{Zero}(\mathbb{P}) = \{(a_1, \cdots, a_n), a_i \in \mathbb{F}_2, s.t., \forall f_i \in \mathbb{P}, f_i(a_1, \cdots, a_n) = 0\}.$$

Let $\mathbb{Q} = \cup_i \mathbb{P}_i$ be the union of some polynomial sets, we define Zero($\mathbb{Q}$) to be $\cup_i$Zero($\mathbb{P}_i$).

Theorem 1. *Algorithm 1 terminates and returns a solution of the Partial Weighted Max-PoSSo problem.*

Algorithm 1. Incremental Solving and Backtracking Search(ISBS) algorithm

```
Input:   Two boolean polynomial sets $\mathcal{H} = \{h_1, h_2, \ldots, h_r\}, \mathcal{S} = \{f_1, f_2, \ldots, f_m\}$.
         A cost function $\mathcal{C}(f_i, x)$,
         where $\mathcal{C}(f_i, x) = 0$ if $f_i(x) = 0$, $\mathcal{C}(f_i, x) = c_i$ if $f_i(x) = 1$.
Output:  $(x_1, \ldots, x_n) \in \mathbb{F}_2^n$ s.t. $h_i(x) = 0$ for any $i$ and $\sum_{f_i \in \mathcal{S}} \mathcal{C}(f_i, x)$ is minimized.
1. Let $E = [e_1, e_2, \ldots, e_m]$ be a m-dim vector.
   Solve $\mathcal{H}$ by an algebraic method and set $\mathbb{Q}_0 = \text{Result}(\mathcal{H})$.
2. For $i$ from 1 to $m$ do
   1.1. Set $\mathbb{P}_i = \{\mathbb{Q}_{i-1}, f_i\}$, solve $\mathbb{P}_i$ and achieve Result($\mathbb{P}_i$).
   1.2. If Result($\mathbb{P}_i$) $= \{1\}$, then set $\mathbb{Q}_i = \mathbb{Q}_{i-1}$ and $e_i = 1$.
   1.3. Else, set $\mathbb{Q}_i = \text{Result}(\mathbb{P}_i)$, $e_i = 0$.
3. Set $\mathbb{S} = \mathbb{Q}_m$ and $u_{bound} = \sum_i c_i e_i$.
   Set $u = u_{bound}$, $k = m$.
4. while $k \geq 1$ do
   4.1. If $e_k = 0$ and $u + c_k < u_{bound}$, then
        4.1.1. Set $e_k = 1$, $u = u + c_k$.
               Solve $\mathbb{P}_k = \{\mathbb{Q}_{k-1}, f_k + 1\}$ and achieve Result($\mathbb{P}_k$).
        4.1.2. If Result($\mathbb{P}_k$) $= \{1\}$, then goto Step 4.2.
        4.1.3. Else, Set $\mathbb{Q}_k = \text{Result}(\mathbb{P}_k)$.
               For $i$ from $k + 1$ to $m$ do
               4.1.3.1. Solve $\mathbb{P}_i = \{\mathbb{Q}_{i-1}, f_i\}$ and achieve Result($\mathbb{P}_i$).
               4.1.3.2. If Result($\mathbb{P}_i$) $= \{1\}$ then $\mathbb{Q}_i = \mathbb{Q}_{i-1}, e_i = 1$.
                        $u = u + c_i$. If $u \geq u_{bound}$, then set $k = i$, and goto Step 4.2.
               4.1.3.3. Else, $\mathbb{Q}_i = \text{Result}(\mathbb{P}_i)$, $e_i = 0$.
        4.1.4. Set $k = m$, $\mathbb{S} = \mathbb{Q}_m$, $u_{bound} = u$.
   4.2. Else, $u = u - e_k c_k$, $k = k - 1$.
5. Get $(x_1, \ldots, x_n)$ from $\mathbb{S}$, and return $(x_1, \ldots, x_n)$.
```

Proof: The termination of Algorithm 1 is trivial, since in this algorithm we are searching all possible values of $[e_1, e_2, \ldots, e_m]$ with backtracking and the number of possible values is finite. In the following, we will explain the operations of this algorithm more specifically, from which we can show the correctness of this algorithm.

From Step 1 to Step 3, we do the following operations recursively. For k from 0 to $m - 1$, We compute Result($\{\mathbb{Q}_k, f_{k+1}\}$).

- If the result is not $\{1\}$, it means that $f_{k+1} = 0$ can be satisfied. We set $e_{k+1} = 0$ and use $\mathbb{Q}_k$ to store the result. Then we compute Result($\{\mathbb{Q}_{k+1}, f_{k+2}\}$).
- If the result is $\{1\}$, $f_{k+1} = 0$ cannot be satisfied by any point in Zero($\mathbb{Q}_k$). It implies that $f_{k+1} = 1$ holds for all these points, so we set $e_{k+1} = 1$. We set $\mathbb{Q}_{k+1} = \mathbb{Q}_k$, and it is also equal to Result($\{\mathbb{Q}_k, f_k + 1\}$). Then we compute Result($\{\mathbb{Q}_{k+1}, f_{k+2}\}$).

After Step 3, we achieve $\mathbb{S}$, and the points in Zero($\mathbb{S}$) are candidates of the problem. For any k with $e_k = 0$, the corresponding polynomial f_k vanishes on Zero($\mathbb{S}$). For any other f_k, $f_k = 0$ is unsatisfied by these points. Note that for the points in Zero($\mathbb{S}$), the values of the corresponding cost functions are same. We use u_{bound} to store this value. Obviously, u, the value of the cost function for a better candidate, should satisfy $u < u_{bound}$.

In Step 4, we are trying to find a better candidate by backtracking to e_k. From $k = m$ to 1, we try to flip the value of e_k, and there are four cases.

1) $e_k = 1$, and Zero($\{f_1 + e_1, \dots, f_{k-1} + e_{k-1}, f_k\}$) $= \emptyset$. We don't flip e_k, since Zero($\{f_1 + e_1, \dots, f_{k-1} + e_{k-1}, f_k\}$) $= \emptyset$ and we can not find any candidate from Zero($\{f_1 + e_1, \dots, f_k\}$).
2) $e_k = 1$, and e_k has already been flipped with the same $e_1, \dots, e_{k-1}$. We don't flip e_k, since we have already considered the points in Zero($\{f_1 + e_1, \dots, f_k\}$).
3) $e_k = 0$, and $u \geq u_{bound}$, where u is the value of the cost function corresponding to $[e_1, \dots, e_{k-1}, e_k + 1]$. We don't flip e_k, since the points in Zero($\{f_1 + e_1, \dots, f_k + e_k + 1\}$) are worse than the stored candidate.
4) $e_k = 0$, and $u < u_{bound}$, where u is the value of the cost function corresponding to $[e_1, \dots, e_{k-1}, e_k + 1]$. We flip e_k.

After we flipping e_k to 1, we compute Result($\{\mathbb{Q}_{k-1}, f_k + 1\}$). This means that we are trying to find better candidates from the points in Zero($\{f_1 + e_1, \dots, f_{k-1} + e_{k-1}, f_k + 1\}$). If Result($\{\mathbb{Q}_{k-1}, f_k + 1\}$) $= \{1\}$, it implies that $\{f_1 + e_1 = 0, \dots, f_{k-1} + e_{k-1} = 0, f_k + 1 = 0\}$ is unsatisfied by any points in $\mathbb{F}_2^n$. Obviously, we cannot find better candidates in this case. When Result($\{\mathbb{Q}_{k-1}, f_k + 1\}$) $\neq \{1\}$, we execute Step 4.1.3.1-4.1.3.3 to incrementally solve the following polynomials $\{f_{k+1}, \dots, f_m\}$ as the operations in Step 1-3. In this procedure, if the value of the cost function u corresponding to $[e_1, \dots, e_i]$ is not smaller than u_{bound}, we have to interrupt the incrementally solving procedure, and backtrack to e_{i-1}. If we successfully complete the incrementally solving process, we will achieve a better candidate $\mathbb{Q}_n$. Then we replace the old $\mathbb{S}$ by $\mathbb{Q}_n$ and refresh the value of u_{bound}. After these operations, we return to Step 4 to backtrack and search again.

From the above statement, we can know that before Step 5 we exhaustively search all the possible values of $[e_1, e_2, \dots, e_m]$ and solve the corresponding polynomial systems $\{f_1 + e_1, f_2 + e_2, \dots, f_m + e_m\}$ in order to find the optimal candidate. We only cut off some branches in the following cases, and we will prove that we cannot achieve a better candidate in these cases .

(a) We obtain $\{1\}$ when incrementally solving $\{f_1 + e_1, \dots, f_{k-1} + e_{k-1}, f_k\}$. In this case, Zero($\{f_1 + e_1, \dots, f_{k-1} + e_{k-1}, f_k\}$ $= \emptyset$ and we cannot find any solution from Zero($\{f_1 + e_1, \dots, f_{k-1} + e_{k-1}, f_k, f_{k+1} + e_{k+1}, \dots, f_m + e_m\}$, where $e_1, \dots, e_{k-1}$ are fixed and $e_{k+1}, \dots, e_m$ can be any values. Thus, we cut off these branches and only consider the branches with $e_k = 1$.
(b) In Step 4.1.3.2, $u \geq u_{bound}$. This implies that the candidates in Zero($\{f_1 + e_1, \dots, f_{i-1} + e_{i-1}, f_i + 1\}$ will not be better than the stored one. Thus, we cut off the following branches and backtrack to e_{i-1}.
(c) In Step 4.1.2, Result($\mathbb{P}_k$) $= \{1\}$. This means that Zero($\{f_1 + e_1, \dots, f_{k-1} + e_{k-1}, f_k + 1\}$) $= \emptyset$. Similarly as case (a), we cut off the following branches because we cannot find any solution from them.

(d) In Step 4.1, when we want to flip e_k from 0 to 1, we find $u + c_k \geq u_{bound}$. This case is similar as case (b). $u + c_k \geq u_{bound}$ implies that the candidates in Zero($\{f_1 + e_1, \ldots, f_{k-1} + e_{k-1}, f_k + 1\}$ will not be better than the stored one, thus we cut off the following branches and backtrack to e_{k-1}.

The loop of Step 4 ends when $k = 0$ which means that we have exhaustively searched all possible branches except the redundant ones and the candidate we stored is the best one among all the points in $\mathbb{F}_2^n$.

Finally, we obtain solutions from $\mathbb{S}$ by Step 5, and this procedure is very easy when $\mathbb{S}$ is some triangular sets or a Gröbner Basis. In most time when $m > n$, S has very simple structure which only contain several points. $\square$

Remark 1. Step 4.1.1 can be changed in Algorithm 1. We only need to set $e_k = 1$, $u = u + c_k$, $\mathbb{Q}_k = \mathbb{Q}_{k-1}$ without solving $\{\mathbb{Q}_{k-1}, f_k + 1\}$. This means that we skip the polynomial f_k in the incremental solving process. Actually, if we achieve a better candidate from the points in Zero($\mathbb{Q}_k$) in the following process, all points in this candidate must satisfy $f_k + 1 = 0$. Suppose $f_k = 0$ for some point in this candidate, then we have $f_1 + e_1 = 0, \ldots, f_{k-1} + e_{k-1} = 0, f_k = 0, f_{k+1} + e_{k+1} = 0, \ldots, f_m + e_m = 0$ hold on this point. Since $f_k = 0$, this point should already be contained in a candidate of the previous process. However, this point is from a better candidate, which means that it is better than itself and leads to a contradiction. This proves the correctness of our modification. Intuitively, after this modification, the algorithm will be more efficient since we don't need to compute Result($\{\mathbb{Q}_{k-1}, f_k + 1\}$), but the opposite is true. From experiments we found that this modification will reduce the efficiency of our algorithm. The reason is that constraint $f_k + 1 = 0$ makes the point sets considered smaller which will accelerate the following compute. More importantly, if Result($\mathbb{P}_k$) = $\{1\}$ we can cut off this backtracking branch instantly. If we consider solving Max-PoSSo over a big finite field, this modification may have some advantage, but this is beyond the scope of this article.

Theoretically estimating the complexity of **ISBS** is very difficult. The only thing we know is that the number of paths in the whole search tree is bounded by 2^m. However, when contradictions occur in the algebraic solving process, a lot of subtree will be cut off. Thus, in our experiments the numbers of paths are much smaller than 2^m.

3.2 The Characteristic Set Method in $\mathbb{F}_2$

It is easy to see that the efficiency of the algebraic solving algorithm will significantly influence the efficiency of the whole algorithm. In our implementation of the **ISBS** method, we use the **MFCS** algorithm as the algebraic solving algorithm. The **MFCS** algorithm is an variant of the Characteristic Set(CS) method for solving the boolean polynomial systems, and it is efficient in the case of incrementally solving. In this subsection, we will simply introduce the **MFCS** algorithm. More details of it can be found in [6].

For a boolean polynomial $P \in \mathbb{F}_2[x_1, x_2, \ldots, x_n]/ < x_1^2 + x_1, x_2^2 + x_2, \ldots, x_n^2 + x_n >$, the *class* of P, denoted as $\mathrm{cls}(P)$, is the largest index c such that x_c occurs in P. If P is a constant, we set $\mathrm{cls}(P)$ to be 0. If $\mathrm{cls}(P) = c > 0$, we call x_c the *leading variable* of P, denoted as $\mathrm{lvar}(P)$. The leading coefficient of P as a univariate polynomial in $\mathrm{lvar}(P)$ is called the *initial* of P, and is denoted as $\mathrm{init}(P)$.

A sequence of nonzero polynomials

$$\mathcal{A}: \quad A_1, A_2, \ldots, A_r \tag{1}$$

is a *triangular set* if either $r = 1$ and $A_1 = 1$ or $0 < \mathrm{cls}(A_1) < \cdots < \mathrm{cls}(A_r)$. A boolean polynomial P is called *monic*, if $\mathrm{init}(P) = 1$. Moreover, if the elements of a triangular set are all monic, we call it a monic triangular set.

Algorithm 2. MFCS($\mathbb{P}$)

Input: A finite set of polynomials $\mathbb{P}$.
Output: Monic triangular sets $\{\mathcal{A}_1, \mathcal{A}_2, \ldots, \mathcal{A}_t\}$ such that $\mathrm{Zero}(\mathbb{P}) = \cup_{i=1}^{t}\mathrm{Zero}(\mathcal{A}_i)$ and $\mathrm{Zero}(\mathcal{A}_i) \cap \mathrm{Zero}(\mathcal{A}_j) = \emptyset$

```
1 Set $\mathbb{P}^* = \{\mathbb{P}\}$, $\mathcal{A}^* = \emptyset$ and $\mathcal{C}^* = \emptyset$.
2 While $\mathbb{P}^* \neq \emptyset$ do
    2.1 Choose a polynomial set $\mathbb{Q}$ from $\mathbb{P}^*$.
    2.2 While $\mathbb{Q} \neq \emptyset$ do
        2.2.1 If $1 \in \mathbb{Q}$, $\mathrm{Zero}(\mathbb{Q}) = \emptyset$. Goto Step 2.1.
        2.2.2 Let $\mathbb{Q}_1 \subset \mathbb{Q}$ be the polynomials with the highest class.
        2.2.3 Let $\mathbb{Q}_{monic} = \emptyset$, $\mathbb{Q}_2 = \mathbb{Q} \setminus \mathbb{Q}_1$.
        2.2.4 While $\mathbb{Q}_1 \neq \emptyset$ do
            Let $P = Ix_c + U \in \mathbb{Q}_1$, $\mathbb{Q}_1 = \mathbb{Q}_1 \setminus \{P\}$.
            $\mathbb{P}_1 = \mathbb{Q}_{monic} \cup \mathbb{Q}_2 \cup \mathbb{Q}_1 \cup \{I, U\}$.
            $\mathbb{P}^* = \mathbb{P}^* \cup \{\mathbb{P}_1\}$.
            $\mathbb{Q}_{monic} = \mathbb{Q}_{monic} \cup \{x_c + U\}$, $\mathbb{Q}_2 = \mathbb{Q}_2 \cup \{I + 1\}$.
        2.2.5 Let $Q = x_c + U$ be a polynomial with lowest degree in $\mathbb{Q}_{monic}$.
        2.2.6 $\mathcal{A} = \mathcal{A} \cup \{Q\}$.
        2.2.7 $\mathbb{Q} = \mathbb{Q}_2 \cup \{R \neq 0 | R = Q_i + Q, Q_i \in \mathbb{Q}_{monic}\}$.
    2.3 if $\mathcal{A} \neq \emptyset$, set $\mathcal{A}^* = \mathcal{A}^* \cup \{\mathcal{A}\}$.
3 Return $\mathcal{A}^*$
```

With the **MFCS** algorithm, we can decompose $\mathrm{Zero}(\mathbb{P})$, the common zero set of a polynomial set $\mathbb{P}$, as $\cup_i \mathrm{Zero}(\mathcal{A}_i)$, the union of the common zero sets of some monic triangular sets $\mathcal{A}_i$. We first convert all polynomials into monic ones by the following decomposition formula: $\mathrm{Zero}(Ix_c + U) = \mathrm{Zero}(Ix_c + U, I + 1) \cup \mathrm{Zero}(I, U) = \mathrm{Zero}(x_c + U, I + 1) \cup \mathrm{Zero}(I, U)$. Note that, with decomposition we will generate some new polynomial sets, and we call these new polynomial sets to be new components. Then we can choose one monic polynomial $x_c + R$ to eliminate the x_c of other polynomials by doing addition $x_c + R + x_c + R_1 = R + R_1$. Note that $R + R_1$ is a polynomial with lower class. Therefore, we can obtain the following group of polynomial sets $\{x_c + R, \mathbb{P}'\}, \mathbb{P}_1, \ldots, \mathbb{P}_t$, where $\mathbb{P}'$ is a set of polynomials with class lower than c and each $\mathbb{P}_i$ is a new generating polynomial set. Then we can recursively apply the above operations to the polynomials with highest class in $\mathbb{P}'$. After dealing with all classes, we will obtain a monic triangular set or constant 1, and generate a group of new polynomial sets. Then we recursively apply the above operations to every new set. Finally, we will obtain the monic triangular sets we need. Obviously, for a monic triangular set $\{x_1 + c_1, x_2 + f_1(x_1), x_3 + f_2(x_2, x_1), \ldots, x_n + f_{n-1}(x_{n-1}, \ldots, x_1)\}$, we can easily solve it.

MFCS has the following properties[6]:

1. The size of polynomials occurring in the whole algorithm can be controlled by that of the input ones. The expansion of the internal result will not happen. Note that in different components most polynomials are same, and the same ones can be shared in the memory with data structure SZDD[11]. For the above reasons, the memory cost of **MFCS** is small.
2. **MFCS** can solve one component very fast. The bitwise complexity of solving one component is $O(LMn^{d+2})$, where L is the number of input polynomials, n is the number of variables, d is the highest degree of the input polynomials and M is the maximal number of terms for all input polynomials. Obviously, when d is fixed, this is a polynomial about n.

4 Experimental Results for Attacking AES and Serpent

According to Section 2.2 we can model the Cold Boot key recovery problems of AES and Serpent as Partial Weighted Max-PoSSo problems. We applied the **ISBS** method to solve these problems and compared our results with those shown in [1]. As in [1], we focused on the 128-bit versions of the two ciphers.

The benchmarks are generated the same way as those in [1]. The experimental platform is a PC with i7 2.8Ghz CPU(only one core is used), and 4G Memory. For every instance of the problem we performed 100 experiments with randomly generated keys, and we only used a reduced round of key schedule. In the first two groups of experiments, we also set δ_1 to be 0.001 as [1,8,12] and used the "aggressive" modelling in most time, where we assume $\delta_1 = 0$ instead of $\delta_1 = 0.001$. In the "aggressive" modelling, all equations with $K_i' = 1$ should be satisfied, so they should be added into the set $\mathcal{H}$ and the problem reduces to Partial Max-PoSSo. Note that, the input data in our experiments are generated with $\delta_1 > 0$, thus in "aggressive" modelling the equations in $\mathcal{H}$ with $K_i = 1$ may be incorrect.

In the following tables, the line "**ISBS**" shows the results of attacking AES and Serpent by using **ISBS**. The line "**SCIP**" shows the results in [1] where they used the **MIP** solver **SCIP** for solving these problems. The column "aggr" denotes whether we choose the aggressive ("+") or normal ("-") modelling. As in [1], we also interrupted the solver when the running time exceeded the time limit. The column "r" gives the success rate, which is the percentage of the instances we recovered the correct key. There are two cases in which we cannot recover the correct key.

– We interrupted the solver after the time limit.
– The optimal solution we achieved from the (Partial weighted) Max-PoSSo problems is not the correct key. In "aggressive" modelling, when some polynomial in $\mathcal{H}$ is incorrect, this will always happen. When all polynomials in $\mathcal{H}$ are correct, if we added polynomials in $\mathbb{F}_{\mathcal{E}}$ which are the checking polynomials into the set $\mathcal{H}$, the optimal solution will always be the correct key. However, as mentioned before we didn't do this in order to decrease the running time. Thus, with a quite low probability, the optimal solution may not be the correct key.

Table 1. AES considering N rounds of key schedule output

δ_0	Method	N	aggr	limit t	r	min t	avg. t	max t
0.15	**ISBS**	4	+	60.0 s	75%	0.002 s	0.07 s	0.15 s
	SCIP	4	+	60.0 s	70%	1.78 s	11.77 s	59.16 s
0.30	**ISBS**	4	+	3600.0 s	70%	0.002 s	0.14 s	2.38 s
	SCIP	4	+	3600.0 s	69%	4.86 s	117.68 s	719.99 s
0.35	**ISBS**	4	+	3600.0 s	66%	0.002 s	0.27 s	7.87 s
	SCIP	4	+	3600.0 s	68%	4.45 s	207.07 s	1639.55 s
0.40	**ISBS**	4	+	3600.0 s	58%	0.002 s	0.84 s	20.30 s
	SCIP	4	+	3600.0 s	61%	4.97 s	481.99 s	3600.00 s
	SCIP	5	+	3600.0 s	62%	7.72 s	704.33 s	3600.00 s
0.50	**ISBS**	4	+	3600.0 s	23%	0.002 s	772.02 s	3600.00 s
	ISBS	5	+	3600.0 s	63%	0.003 s	1.05 s	46.32 s
	SCIP	4	+	3600.0 s	8%	6.57 s	3074.36 s	3600.00 s
	SCIP	4	+	7200.0 s	13%	6.10 s	5882.66 s	7200.00 s

Table 1 presents the results of attacking AES. For attacking AES, we didn't use the normal("-") modelling.[3] In the aggressive modelling, by adding some intermediate variables we convert the S-box polynomials with degree 7 into the quadratic polynomials[3]. Since these intermediate quadratic polynomials must be satisfied, we add them into set $\mathcal{H}$. For these problems from AES, after we solving $\mathcal{H}$, the point in Zero(Result($\mathcal{H}$)) is small, finding an optimal one from this set is not too hard. Thus, most of the running time is spent on solving $\mathcal{H}$. When we use more rounds of key schedule output, we can solve $\mathcal{H}$ faster. This explains why the result of $N = 5$ is better than that of $N = 4$ in the case of $\delta_0 = 0.5$. From the results of **SCIP**, it seems that more rounds will lead to a worse result. From the results of Table 1, we can see that, for the easy problems, with keeping the same success rate, the running time of **ISBS** is much shorter.

[3] Since the first round key of AES is its initial key, we have 128 polynomials which have form $x_i + 1$ or x_i. In this case, using **ISBS** to search the value of polynomials is equal to exhaustively searching the value of variables.

For the hard problem, the success rate of **ISBS** is higher, and the running time of it is shorter.

In the aspect of attacking cipher, our results are worse than those in [12]. Actually, as mentioned before, if we use all rounds of the key schedule, the running times of **ISBS** will be much shorter and close to the running times in [12]. However, using all rounds will make Zero(Result($\mathcal{H}$)) only contain one point, and this means that we just need to solve a PoSSo problem instead of a Partial Max-PoSSo problem. Hence, for testing the efficiency of **ISBS** for solving Partial Max-PoSSo problems, we only use 4 or 5 rounds of key schedule and achieved these poorer attack results.

Table 2. SERPENT considering $32 \cdot N$ bits of key schedule output

δ_0	Method	N	aggr	limit t	r	min t	avg. t	max t
0.05	**ISBS**	8	−	600.0 s	90%	0.41 s	58.49 s	600.00 s
	SCIP	12	−	600.0 s	37%	8.22 s	457.57s	600.00 s
0.15	**ISBS**	12	+	60.0 s	81%	1.19 s	3.82 s	60.00 s
	SCIP	12	+	60.0 s	84%	0.67 s	11.25 s	60.00 s
	SCIP	16	+	60.0 s	79%	0.88 s	13.49 s	60.00 s
0.30	**ISBS**	16	+	600.0 s	81%	4.73 s	11.66 s	58.91 s
	SCIP	16	+	600.0 s	74%	1.13s	57.05 s	425.48 s
0.50	**ISBS**	20	+	3600.0 s	55%	14.11 s	974.69 s	3600.00 s
	SCIP	16	+	3600.0 s	38%	136.54 s	2763.68 s	3600.00 s

The results of attacking Serpent are given in Table 2. In the normal modelling with $\delta_0 = 0.05$, we set $N = 8$. The reason is that in this modelling we don't have polynomials in $\mathcal{H}$, and more input polynomials will make us search more possible values of these polynomials. If $N = 4$, we can get an optimal result very fast, but it isn't the correct key in most time. The reason is that the randomness of the first round of the Serpent key schedule is poor. By setting $N = 8$, we have a good success rate and short running times. In the aggressive modelling, the situations are similar to those in AES. For **SCIP**, when N is larger, the results are worse. For **ISBS**, more bits of key schedule will make it solve $\mathcal{H}$ easier. Therefore, when δ_0 increase we set N larger.

Table 3 presents the results when considering symmetric noise(i.e., $\delta_0 = \delta_1$). This is a pure Max-PoSSo problem. As mentioned before, when $\mathcal{H} = \emptyset$, with less equations **ISBS** can be more efficient. Thus, in this group of experiments, we set $N = 8$. When solving these problems, we used some important tricks to accelerate the computation. These tricks will be introduced in the appendix detailedly. From the results, we can see that **ISBS** has higher success rates and shorter running time comparing to **SCIP**.

If we don't use the incremental solving and backtracking search method, which means that we only exhaustively search the value of polynomials in the order of increasing Hamming-weight and solve the corresponding polynomial systems, when $\delta_0 = \delta_1 = 0.05$ the running time will be about $\binom{256}{0.05 \cdot 256} \cdot T_0 \approx 2^{71} \cdot$

Table 3. SERPENT considering $32 \cdot N$ bits of key schedule output(symmetric noise)

$\delta_0 = \delta_1$	Method	N	limit t	r	min t	avg. t	max t
0.01	**ISBS**	8	3600.0 s	100%	0.78 s	9.87 s	138.19 s
	SCIP	12	3600.0 s	96%	4.60 s	256.46 s	3600.00 s
0.02	**ISBS**	8	3600.0 s	96%	0.80 s	163.56 s	3600.00 s
	SCIP	12	3600.0 s	79%	8.20 s	1139.72 s	3600.00 s
0.03	**ISBS**	8	3600.0 s	90%	1.74 s	577.60 s	3600.00 s
	SCIP	12	7200.0 s	53%	24.57 s	4205.34 s	7200.0 s
0.05	**ISBS**	8	3600.0 s	38%	12.37 s	917.05 s	3600.00 s
	SCIP	12	3600.0 s	18%	5.84 s	1921.89 s	3600.00 s

T_0, where T_0 is the time for solving a polynomial system with 128 variables and 256 equations. This polynomial system can be easily solved, because of the easy invertibility of the key schedule operations. From our experiments with 100 instances, the average value of T_0 is 0.30 seconds, then $2^{71} \cdot T_0 \approx 2^{69.3}$ seconds which is much larger than our result $917.05 \approx 2^{9.8}$ seconds.[4] This implies that by using incremental solving and backtracking search method, we avoid a lot of repeated computation and cut off a lot of redundant branches which highly improve the efficiency of searching.

5 Conclusion

In this paper, we proposed a new method called **ISBS** for solving the family of Max-PoSSo problems over $\mathbb{F}_2$, and applied the method in solving the Cold Boot key recovery problems of AES and Serpent. Our work is inspired by the work in [1], and provide a new way of solving the non-linear polynomials system with noise. Our method was combined with the Characteristic Set method, which is a powerful tool in symbolic computation and has good performances on solving the boolean polynomial systems. The main innovation of **ISBS** is the combination of incremental solving and backtracking search. By this idea, we can use the former results to reduce the repeated computations and use the conflictions to cut off a lot of redundant branches. Our experimental data shows that **ISBS** has good performances on solving the Cold Boot key recovery problems of AES and Serpent, and its results are better than the previously existing ones by using **SCIP** solver.

References

1. Albrecht, M., Cid, C.: Cold Boot Key Recovery by Solving Polynomial Systems with Noise. In: Lopez, J., Tsudik, G. (eds.) ACNS 2011. LNCS, vol. 6715, pp. 57–72. Springer, Heidelberg (2011)

[4] With the tricks introduced in appendix, for every instance in our experiments, we can search and solve all branches by **ISBS** within the time limit under the condition that the number of unsatisfied polynomials $\leq 0.05 \cdot 256 \approx 13$.

2. Chai, F., Gao, X.S., Yuan, C.: A Characteristic Set Method for Solving Boolean Equations and Applications in Cryptanalysis of Stream Ciphers. Journal of Systems Science and Complexity 21(2), 191–208 (2008)
3. 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)
4. Faugère, J.C.: A New Efficient Algorithm for Computing Gröbner Bases (F4). Journal of Pure and Applied Algebra 139(1-3), 61–88 (1999)
5. Faugère, J.C.: A New Efficient Algorithm for Computing Gröner Bases Without Reduction to Zero (F5). In: Proc. ISSAC, pp. 75–83 (2002)
6. Gao, X.S., Huang, Z.: Efficient Characteristic Set Algorithms for Equation Solving in Finite Fields. Journal of Symbolic Computation 47(6), 655–679 (2012)
7. Håstad, J.: Satisfying Degree-d Equations over $GF[2]^n$. In: Goldberg, L.A., Jansen, K., Ravi, R., Rolim, J.D.P. (eds.) APPROX/RANDOM 2011. LNCS, vol. 6845, pp. 242–253. Springer, Heidelberg (2011)
8. 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. In: USENIX Security Symposium, pp. 45–60. USENIX Association (2009)
9. Huang, Z., Lin, D.: Attacking Bivium and Trivium with the Characteristic Set Method. In: Nitaj, A., Pointcheval, D. (eds.) AFRICACRYPT 2011. LNCS, vol. 6737, pp. 77–91. Springer, Heidelberg (2011)
10. Huang, Z.: Parametric Equation Solving and Quantifier Elimination in Finite Fields with the Characteristic Set Method. Jounral of Systmes Science and Complexity 25(4), 778–791 (2012)
11. Minto, S.: Zero-Sppressed BDDs for Set Manipulation in Combinatorial Problems. In: Proc. ACM/IEEE Design Automation, pp. 272–277. ACM Press (1993)
12. Kamal, A.A., Youssef, A.M.: Applications of SAT Solvers to AES key Recovery from Decayed Key Schedule Images. In: Proceedings of the Fourth International Conference on Emerging Security Information, Systems and Technologies, SECURWARE 2010, Venice/Mestre, Italy, July 18-25 (2010)
13. Wu, W.T.: Basic Principles of Mechanical Theorem-proving in Elementary Geometries. Journal Automated Reasoning 2, 221–252 (1986)
14. Zhao, S.W., Gao, X.S.: Minimal Achievable Approximation Ratio for MAX-MQ in Finite Fields. Theoretical Computer Science 410(21-23), 2285–2290 (2009)

Appendix: Some Tricks Used in ISBS for Solving Symmetric Noisy Problems

In this section, we will introduce some tricks we used in solving the problems in Table 3. For this kind of problems, these tricks can greatly improve the efficiency of **ISBS**.

(I) In **ISBS**, if u_{bound} is smaller, the branches we need to search is less, and the algorithm will end faster. When u_{min}, the value of the cost function corresponding to the optimal solution, is small, we can increasingly set the u_{bound} as $k, 2k, 3k, \ldots, u_0$. Here k is a value which is determined by the number of input polynomials, and u_0 is the value of the cost function corresponding to $\mathbb{Q}_n$ which is generated from Step 2. Then we try to find a solution under these bounds. Precisely speaking, in Step 3, we set $\mathbb{S} = \emptyset$ and $u_{bound} = mk$, $m = 1, 2, \dots$ If **ISBS** returns a nonempty solution when $u_{bound} = mk < u_0$ for some m, then it is the optimal solution of the problem. Otherwise, we set $u_{bound} = (m+1)k$ if $(m+1)k < u_0$; or $u_{bound} = u_0$ if $(m+1)k \geq u_0$. Then we execute Step 3-5 of **ISBS** again until we achieve a nonempty solution. If we cannot achieve a nonempty solution, $\mathbb{Q}_n$ from Step 2 will be the optimal solution. The disadvantage of this modification is that in the case of $u_{bound} = (m+1)k$ we need to search a lot of branches which have already be searched in the case of $u_{bound} = mk$. If u_{min} is big, the cost of repeated computation is big. That is why we only use this modification when u_{min} is small.

(II) Note that, in our experiments, we used 256 bits of key schedule output, while the key is 128 bits. Thus the input polynomial system has 128 variables and 256 polynomials. In the experiments, after we had incrementally solved the first 128 polynomials with any assignment of $[e_1, \dots, e_{128}]$, we always obtained a result which only contains several points. Then the process of solving and backtracking the following 128 polynomials is very easy. For example, if $\text{Zero}(\{f_1 + e'_1, \dots, f_{128} + e'_{128}\}) = \{x_0\}$ for some $[e'_1, \dots, e'_{128}]$, where x_0 is a point in $\mathbb{F}_2^n$. Then $[e_{129}, \dots, e_{256}]$ must be equal to $[f_{129}(x_0), \dots, f_{256}(x_0)]$. Any flip of $e_i, i = 129, \dots, 256$ will lead to $\text{Result}(\text{Result}(\{f_1 + e'_1, \dots, f_{128} + e'_{128}\}), f_i + e_i + 1) = 1$, so this branch will be cut off instantly.

Based on the above observation, given a u_{bound}, the running time of **ISBS** is almost equal to the time of searching and solving $\{f_1 + e_1, f_2 + e_2, \dots, f_{128} + e_{128}\}$ where $[e_1, \dots, e_{128}]$ satisfies $u(e_1, e_2, \dots, e_{128}) \leq u_{bound}$ and $u(e_1, e_2, \dots, e_{128})$ is the value of the cost function corresponding to $[e_1, e_2, \dots, e_{128}]$. The cost of the operations about the following 128 polynomials can be ignored.

For an assignment $[e_1^0, e_2^0, \dots, e_{256}^0]$ of $[e_1, e_2, \dots, e_{256}]$ satisfying $u(e_1^0, e_2^0, \dots, e_{256}^0) \leq u_{bound}$, we have $u(e_1^0, \dots, e_{128}^0) \leq \frac{1}{2}u_{bound}$ or $u(e_{129}^0, \dots, e_{256}^0) \leq \frac{1}{2}u_{bound}$. Instead of searching all the branches satisfying $u(e_1, e_2, \dots, e_{128}) \leq u_{bound}$, we can search the branches satisfying $u(e_1, e_2, \dots, e_{128}) \leq \frac{1}{2}u_{bound}$ or $u(e_{129}, e_{130}, \dots, e_{256}) \leq \frac{1}{2}u_{bound}$. To do this, we can use **ISBS** one time to solve $\{f_1, \dots, f_{128}, f_{129}, \dots, f_{256}\}$ under the condition of $u(e_1, e_2, \dots, e_{128}) \leq \frac{1}{2}u_{bound}$, then use **ISBS** another time to solve $\{f_{129}, \dots, f_{256}, f_1, \dots, f_{128}\}$ under the condition of $u(e_{129}, e_{130}, \dots, e_{256}) \leq \frac{1}{2}u_{bound}$. To understand why this modification can accelerate the computation, we need the following two lemmas.

Lemma 1. *Let a, b, n be positive integers, and $n > a+b$. Then* $\sum_{i=0}^{a}\binom{n}{i} + \sum_{i=0}^{b}\binom{n}{i} \geq \sum_{i=0}^{\lceil\frac{a+b}{2}\rceil}\binom{n}{i} + \sum_{i=0}^{\lfloor\frac{a+b}{2}\rfloor}\binom{n}{i}$. *The equality holds if and only if $a = b$ or $a + b = n - 1$.*

Proof: Obviously, when $a = b$ the equality holds. If $a + b = n - 1$, $\sum_{i=0}^{a}\binom{n}{i} + \sum_{i=0}^{b}\binom{n}{i} = \sum_{i=0}^{a}\binom{n}{i} + \sum_{i=n-b}^{n}\binom{n}{i} = \sum_{i=0}^{a}\binom{n}{i} + \sum_{i=a+1}^{n}\binom{n}{i} = 2^n$. $\sum_{i=0}^{\lceil\frac{a+b}{2}\rceil}\binom{n}{i} + \sum_{i=0}^{\lfloor\frac{a+b}{2}\rfloor}\binom{n}{i} =$

$\sum_{i=0}^{\lceil\frac{n-1}{2}\rceil}\binom{n}{i} + \sum_{i=n-\lfloor\frac{n-1}{2}\rfloor}^{n}\binom{n}{i} = \sum_{i=0}^{\lceil\frac{n-1}{2}\rceil}\binom{n}{i} + \sum_{i=\lceil\frac{n-1}{2}\rceil+1}^{n}\binom{n}{i} = 2^n$. Thus, the equality holds when $a + b = n - 1$.

If $a \neq b$ and $a + b < n - 1$, we can assume $a > b$ without loss of generality. Then it is equal to prove

$$\binom{n}{b+1} + \binom{n}{b+2} + \cdots + \binom{n}{\lfloor\frac{a+b}{2}\rfloor} < \binom{n}{\lceil\frac{a+b}{2}\rceil+1} + \binom{n}{\lceil\frac{a+b}{2}\rceil+2} + \cdots + \binom{n}{a}. \tag{2}$$

Note that both sides of the inequality have the same number of terms.

- If $a \leq \lfloor\frac{n}{2}\rfloor$, obviously we have $\binom{n}{b+1} < \binom{n}{\lceil\frac{a+b}{2}\rceil+1}$, $\binom{n}{b+2} < \binom{n}{\lceil\frac{a+b}{2}\rceil+2}, \ldots,$ $\binom{n}{\lfloor\frac{a+b}{2}\rfloor} < \binom{n}{a}$. Summing up all these inequalities, we can obtain (2).
- If $a > \lfloor\frac{n}{2}\rfloor$, then we can divide the right part of (2) into two parts

$$\binom{n}{\lceil\frac{a+b}{2}\rceil+1} + \cdots + \binom{n}{\lfloor\frac{n}{2}\rfloor}, \binom{n}{\lfloor\frac{n}{2}\rfloor+1} + \cdots + \binom{n}{a}, \tag{3}$$

and also divide the left part of (2) into two parts

$$\binom{n}{a+b-\lfloor\frac{n}{2}\rfloor+1} + \cdots + \binom{n}{\lfloor\frac{a+b}{2}\rfloor}, \binom{n}{b+1} + \cdots + \binom{n}{a+b-\lfloor\frac{n}{2}\rfloor}. \tag{4}$$

The first parts of (3) and (4) have the same number of terms, and the second parts of (3) and (4) also have the same number of terms. Since $\lfloor\frac{n}{2}\rfloor > \lfloor\frac{a+b}{2}\rfloor$, we have $\binom{n}{a+b-\lfloor\frac{n}{2}\rfloor+1} < \binom{n}{\lceil\frac{a+b}{2}\rceil+1}, \ldots, \binom{n}{\lfloor\frac{a+b}{2}\rfloor} < \binom{n}{\lfloor\frac{n}{2}\rfloor}$. Then $\binom{n}{a+b-\lfloor\frac{n}{2}\rfloor+1} + \cdots + \binom{n}{\lfloor\frac{a+b}{2}\rfloor} < \binom{n}{\lceil\frac{a+b}{2}\rceil+1} + \cdots + \binom{n}{\lfloor\frac{n}{2}\rfloor}$. Then it is sufficient to prove that the second part of (3) is not less than that of (4). We have $\binom{n}{\lfloor\frac{n}{2}\rfloor+1} + \binom{n}{\lfloor\frac{n}{2}\rfloor+2} + \cdots + \binom{n}{a} = \binom{n}{\lceil\frac{n}{2}\rceil-1} + \binom{n}{\lceil\frac{n}{2}\rceil-2} \cdots + \binom{n}{n-a} = \binom{n}{n-a} + \binom{n}{n-a+1} + \cdots + \binom{n}{\lceil\frac{n}{2}\rceil-1}$. Since $n - a > b + 1$, we have $\binom{n}{n-a} > \binom{n}{b+1}, \ldots, \binom{n}{\lceil\frac{n}{2}\rceil-1} > \binom{n}{a+b-\lfloor\frac{n}{2}\rfloor}$. This proves the inequality (2). □

Furthermore, we can prove the following lemma similarly as Lemma 1,

Lemma 2. *Let a, b, n be positive integers. Assume $n > a + b$ and $a > b$. Then $\sum_{i=0}^{a}\binom{n}{i} + \sum_{i=0}^{b}\binom{n}{i} \geq \sum_{i=0}^{a-t}\binom{n}{i} + \sum_{i=0}^{b+t}\binom{n}{i}$, where t is an integer satisfying $0 < t < \frac{a-b}{2}$. The equality holds if and only if $a + b = n - 1$.*

According to Lemma 1, we have $\sum_{i=0}^{u_{bound}}\binom{128}{i} > \sum_{i=0}^{\frac{1}{2}u_{bound}}\binom{128}{i} + \sum_{i=0}^{\frac{1}{2}u_{bound}}\binom{128}{i}$. In the problems of Table 3, the value of the cost function is the number of unsatisfying equations. Thus, the inequality implies that the number of branches satisfying

$u(e_1, e_2, \ldots, e_{128}) \leq u_{bound}$ is larger than the number of branches satisfying $u(e_1, e_2, \ldots, e_{128}) \leq \frac{1}{2}u_{bound}$ or $u(e_{129}, e_{130}, \ldots, e_{256}) \leq \frac{1}{2}u_{bound}$ without considering cutting off branching. For example, if $u_{bound} = 10$, then $\sum_{i=0}^{u_{bound}} \binom{128}{i} \approx 2^{47.8}$ and $\sum_{i=0}^{\frac{1}{2}u_{bound}} \binom{128}{i} + \sum_{i=0}^{\frac{1}{2}u_{bound}} \binom{128}{i} \approx 2^{29.0}$. Obviously, this is a remarkable improvement.

However, in the Serpent problems, solving a branch with input $\{f_{129}, \ldots, f_{256}, f_1, \ldots, f_{128}\}$ is slower than solving a branch with input $\{f_1, \ldots, f_{128}, f_{129}, \ldots, f_{256}\}$. The reason is that $f_1, \ldots, f_{256}$ is a polynomials sequence which is sorted from the "simplest" one to the "most complex" one. In this order incremental solving can be more efficient. A better strategy is considering the branches satisfying $u(e_1, e_2, \ldots, e_{128}) \leq \frac{6}{10}u_{bound}$ or $u(e_{129}, e_{130}, \ldots, e_{256}) \leq \frac{4}{10}u_{bound}$. From Lemma 2, we know that under this strategy the branches we need to solve are still much less than those in the case of $u(e_1, e_2, \ldots, e_{128}) \leq u_{bound}$. From experiments, we found that the total running time under this strategy will be smaller than that under the strategy of considering $u(e_1, e_2, \ldots, e_{128}) \leq \frac{1}{2}u_{bound}$ or $u(e_{129}, e_{130}, \ldots, e_{256}) \leq \frac{1}{2}u_{bound}$.

In the experiments of Table 3 when $\delta_0 = 0.01, 0.02, 0.03$, we used trick (I). For the problems with $\delta_0 = 0.05$, we used both trick (I) and trick (II).

Cryptanalysis of the Xiao – Lai White-Box AES Implementation

Yoni De Mulder[1], Peter Roelse[2], and Bart Preneel[1]

[1] KU Leuven
Dept. Elect. Eng.-ESAT/SCD-COSIC and IBBT,
Kasteelpark Arenberg 10, 3001 Heverlee, Belgium
{yoni.demulder,bart.preneel}@esat.kuleuven.be
[2] Irdeto B.V.
Taurus Avenue 105, 2132 LS Hoofddorp, The Netherlands
peter.roelse@irdeto.com

Abstract. In the white-box attack context, i.e., the setting where an implementation of a cryptographic algorithm is executed on an untrusted platform, the adversary has full access to the implementation and its execution environment. In 2002, Chow *et al.* presented a white-box AES implementation which aims at preventing key-extraction in the white-box attack context. However, in 2004, Billet *et al.* presented an efficient practical attack on Chow *et al.*'s white-box AES implementation. In response, in 2009, Xiao and Lai proposed a new white-box AES implementation which is claimed to be resistant against Billet *et al.*'s attack. This paper presents a practical cryptanalysis of the white-box AES implementation proposed by Xiao *et al.* The linear equivalence algorithm presented by Biryukov *et al.* is used as a building block. The cryptanalysis efficiently extracts the AES key from Xiao *et al.*'s white-box AES implementation with a work factor of about 2^{32}.

Keywords: white-box cryptography, AES, cryptanalysis, linear equivalence algorithm.

1 Introduction

A white-box environment is an environment in which an adversary has complete access to an implementation of a cryptographic algorithm and its execution environment. In a white-box environment, the adversary is much more powerful than in a traditional black-box environment in which the adversary has only access to the inputs and outputs of a cryptographic algorithm. For example, in a white-box environment the adversary can: (1) trace every program instruction of the implementation, (2) view the contents of memory and cache, including secret data, (3) stop execution at any point and run an off-line process, and/or (4) alter code or memory contents at will. To this end, the adversary can make use of widely available tools such as disassemblers and debuggers.

An example of a white-box environment is a digital content protection system in which the client is implemented in software and executed on a PC, tablet, set-top box or a mobile phone. A malicious end-user may attempt to extract a secret key used for content decryption from the software. Next, the end-user may distribute this key to non-entitled end-users, or the end-user may use this key to decrypt the content directly, circumventing content usage rules.

White-box cryptography was introduced in 2002 by Chow, Eisen, Johnson and van Oorschot in [4,5], and aims at protecting a secret key in a white-box environment. In [4], Chow *et al.* present generic techniques that can be used to design implementations of a cryptographic algorithm that resist key extraction in a white-box environment. Next, the authors apply these techniques to define an example white-box implementation of the Advanced Encryption Standard (AES).

In 2004, a cryptanalysis of the white-box AES implementation by Chow *et al.* was presented by Billet, Gilbert and Ech-Chatbi [1]. This attack is referred to as the Billet Gilbert Ech-Chatbi (BGE) attack in the following. The BGE attack is efficient in that a modern PC only requires a few minutes to extract the AES key from the white-box AES implementation. In [7], James Muir presents a tutorial on the design and cryptanalysis of white-box AES implementations.

The BGE attack motivated the design of other white-box AES implementations offering more resistance against key extraction. In [3], Bringer, Chabanne and Dottax proposed a white-box AES implementation in which perturbations are added to AES in order to hide its algebraic structure. However, the implementation in [3] has been cryptanalyzed by De Mulder, Wyseur and Preneel in [8]. Recently, two new white-box AES implementations have been proposed: one in 2010 by Karroumi based on dual ciphers of AES [6] and one in 2009 by Xiao and Lai based on large linear encodings [10].

This paper presents a cryptanalysis of the Xiao – Lai white-box AES implementation proposed in [10], efficiently extracting the AES key from the white-box AES implementation. The cryptanalysis uses the linear equivalence algorithm presented by Biryukov, De Cannière, Braeken and Preneel in [2] as a building block. In addition to this, the structure of AES and the structure of the white-box implementation are exploited in the cryptanalysis. Key steps of the cryptanalysis have been implemented in `C++` and verified by computer experiments.

Organization of This Paper. The remainder of this paper is organized as follows. Section 2 briefly describes the white-box AES implementation proposed in [10] and the linear equivalence algorithm presented in [2]. Section 3 outlines the cryptanalysis of the white-box AES implementation. Finally, concluding remarks can be found in Sect. 4.

2 Preliminaries

2.1 AES-128

In this section, aspects of AES-128 that are relevant for this paper are described. For detailed information, refer to FIPS 197 [9]. AES-128 is an iterated block cipher mapping a 16 byte plaintext to a 16 byte ciphertext using a 128 bit key. AES-128 consists of 10 rounds and has 11 round keys which are derived from the AES-128 key using the AES key scheduling algorithm. Each round of the algorithm updates a 16 byte state; the initial state of the algorithm is the plaintext and the final state of the algorithm is the ciphertext. In the following, a state is denoted by $[\text{state}_i]_{i=0,1,\dots,15}$. A round comprises the following operations:

- *ShiftRows* is a permutation on the indices of the bytes of the state. It is defined by the permutation $(0, 5, 10, 15, 4, 9, 14, 3, 8, 13, 2, 7, 12, 1, 6, 11)$, i.e. the first byte of the output of `ShiftRows` is the first byte of the input, the second byte of the output is the fifth byte of the input, and so on.
- *AddRoundKey* is a bitwise addition modulo two of a 128 bit round key k^r $(1 \leq r \leq 11)$ and the state.
- *SubBytes* applies the AES S-box operation to every byte of the state. AES uses one fixed S-box, denoted by S, which is a non-linear, bijective mapping from 8 bits to 8 bits
- *MixColumns* is a linear operation over GF $\left(2^8\right)$ operating on 4 bytes of the state at a time. The `MixColumns` operation can be represented by a 4×4 matrix `MC` over GF $\left(2^8\right)$. To update the state, 4 consecutive bytes of the state are interpreted as a vector over GF $\left(2^8\right)$ and multiplied by `MC`. Using the notation and the representation of the finite field as in [9], we have:

$$\begin{pmatrix} \text{state}_{4i} \\ \text{state}_{4i+1} \\ \text{state}_{4i+2} \\ \text{state}_{4i+3} \end{pmatrix} \leftarrow \texttt{MC} \cdot \begin{pmatrix} \text{state}_{4i} \\ \text{state}_{4i+1} \\ \text{state}_{4i+2} \\ \text{state}_{4i+3} \end{pmatrix} \quad \text{with} \quad \texttt{MC} = \begin{pmatrix} 02\ 03\ 01\ 01 \\ 01\ 02\ 03\ 01 \\ 01\ 01\ 02\ 03 \\ 03\ 01\ 01\ 02 \end{pmatrix},$$

  for $i = 0, 1, 2, 3$.

There are several equivalent ways to describe AES-128. The following description of AES-128 is the one used in this paper, where $\hat{k}^r$ for $1 \leq r \leq 10$ is the result of applying `ShiftRows` to k^r:

```
state ← plaintext
for $r = 1$ to 9 do
    state ← ShiftRows (state)
    state ← AddRoundKey(state,$\hat{k}^r$)
    state ← SubBytes(state)
    state ← MixColumns (state)
end for
state ← ShiftRows (state)
state ← AddRoundKey(state,$\hat{k}^{10}$)
state ← SubBytes(state)
state ← AddRoundKey(state,$k^{11}$)
ciphertext ← state
```

2.2 The White-Box AES Implementation

This section describes the white-box AES implementation proposed in [10]. As the `MixColumns` operation is omitted in the final AES-128 round, the white-box implementation of the final round differs from the white-box implementation of the other rounds. However, as the final round is not relevant for the cryptanalysis presented in this paper, the description of its implementation is omitted below.

First, the `AddRoundKey` and `SubBytes` operations of AES round r $(1 \leq r \leq 9)$ are composed, resulting in 16 8-bit bijective lookup tables for each round. In the following, such a table is referred to as a T-box. If the 16 bytes of a 128 bit round key are denoted by $\hat{k}_i^r$ $(i = 0, 1, \ldots, 15)$, then the T-boxes are defined as follows:

$$T_i^r(x) = S(x \oplus \hat{k}_i^r) \quad \text{for } 1 \leq r \leq 9 \text{ and } 0 \leq i \leq 15 \ .$$

Second, the 4×4 matrix $\texttt{MC}$ is split into two 4×2 submatrices: $\texttt{MC}_0$ is defined as the first 2 columns of $\texttt{MC}$ and $\texttt{MC}_1$ is defined as the remaining 2 columns of $\texttt{MC}$. Using this notation, the `MixColumns` matrix multiplication is given by:

$$\begin{pmatrix} \text{state}_{4i} \\ \text{state}_{4i+1} \\ \text{state}_{4i+2} \\ \text{state}_{4i+3} \end{pmatrix} \leftarrow \texttt{MC}_0 \cdot \begin{pmatrix} \text{state}_{4i} \\ \text{state}_{4i+1} \end{pmatrix} \oplus \texttt{MC}_1 \cdot \begin{pmatrix} \text{state}_{4i+2} \\ \text{state}_{4i+3} \end{pmatrix}$$

for $i = 0, 1, 2, 3$.

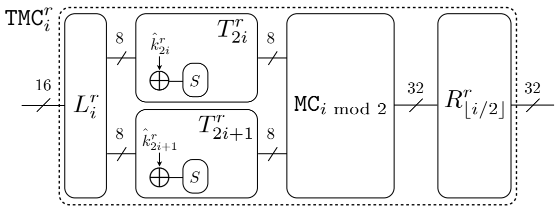


Fig. 1. Composition of T-boxes (T_{2i}^r and T_{2i+1}^r) and `MixColumns` operation ($\texttt{MC}_{i \bmod 2}$) resulting in 16-to-32 bit lookup table $\texttt{TMC}_i^r$

For $1 \leq r \leq 9$, the T-boxes and the `MixColumns` operations are composed as depicted in Fig. 1. Observe that this results in 8 lookup-tables per round, each table mapping 16 bits to 32 bits. To prevent an adversary from extracting the AES round keys from these tables, each table is composed with two secret white-box encodings L_i^r and $R_{\lfloor i/2 \rfloor}^r$ as depicted in Fig. 1. Each white-box encoding L_i^r is a bijective linear mapping from 16 bits to 16 bits, i.e., it can be represented by a non-singular 16×16 matrix over GF (2). Each white-box encoding $R_{\lfloor i/2 \rfloor}^r$ is a bijective linear mapping from 32 bits to 32 bits, i.e., it can be represented by a non-singular 32×32 matrix over GF (2). The resulting tables from 16 to 32 bits are referred to as $\texttt{TMC}_i^r$ $(i = 0, 1, \ldots, 7)$ in the following.

Third, a 128 × 128 non-singular matrix M^r over GF (2) is associated with each round r ($1 \leq r \leq 9$). If SR denotes the 128 × 128 non-singular matrix over GF (2) representing the ShiftRows operation, then the matrix M^r is defined as follows:

$$M^r = \mathrm{diag}\Big((L_0^r)^{-1}, \ldots, (L_7^r)^{-1}\Big) \circ \mathtt{SR} \circ \mathrm{diag}\Big((R_0^{r-1})^{-1}, \ldots, (R_3^{r-1})^{-1}\Big) \ , \quad (1)$$

for $r = 2, 3, \ldots, 9$, where '$\circ$' denotes the function composition symbol. The matrix M^1 associated with the first round has a slightly different structure and is defined below.

Fourth, an additional secret white-box encoding is defined, denoted by IN. This encoding is represented by a non-singular 128 × 128 matrix over GF (2), and is applied to an AES-128 plaintext. Next, the non-singular 128 × 128 matrix M^1 over GF (2) is defined as follows:

$$M^1 = \mathrm{diag}\Big((L_0^1)^{-1}, \ldots, (L_7^1)^{-1}\Big) \circ \mathtt{SR} \circ \mathtt{IN}^{-1} \ . \quad (2)$$

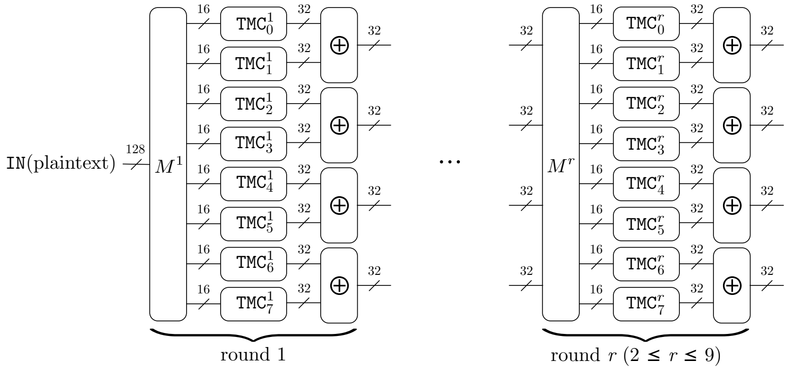

IN(plaintext)
128
M^1
16
$\mathtt{TMC}_0^1$ … $\mathtt{TMC}_7^1$
32
⊕
round 1
…
M^r
$\mathtt{TMC}_0^r$ … $\mathtt{TMC}_7^r$
round r ($2 \leq r \leq 9$)

Fig. 2. White-box AES-128 implementation [10] of rounds $r = 1, 2, \ldots, 9$

Using these notations and definitions, the structure of the first 9 rounds of the white-box AES-128 implementation is depicted in Fig. 2. In the white-box implementation, an operation M^r is implemented as a matrix vector multiplication over GF (2) and an operation $\mathtt{TMC}_i^r$ is implemented as a look-up table. Notice that the output of two tables, which corresponds to the linearly encoded output of $\mathtt{MC}_0$ and $\mathtt{MC}_1$, is added modulo two in the white-box implementation. After the final AES round, a secret white-box encoding OUT is applied to the AES-128 ciphertext. OUT is represented by a non-singular 128 × 128 matrix over GF (2). Observe that, with the exception of the encodings IN and OUT, the white-box implementation of AES-128 is functionally equivalent to AES-128.

The main differences with the white-box AES-128 implementation presented in [4] are the following: (i) all secret white-box encodings are linear over GF (2), and (ii) the secret white-box encodings operate on at least 2 bytes simultaneously instead of at least 4 bits (in case of a non-linear encoding) or at least a byte (in case of a linear encoding) in [4]. In [10], the authors argue that their white-box AES-128 implementation is resistant against the BGE attack [1].

2.3 The Linear Equivalence Algorithm

Definition 1. *Two permutations on n bits (or S-boxes) S_1 and S_2 are called linearly equivalent if a pair of linear mappings (A, B) from n to n bits exists such that $S_2 = B \circ S_1 \circ A$.*

A pair (A, B) as in this definition is referred to as a linear equivalence. Notice that both linear mappings A and B of a linear equivalence are bijective. If $S_1 = S_2$, then the linear equivalences are referred to as linear self-equivalences.

The linear equivalence problem is: given two n-bit bijective S-boxes S_1 and S_2, determine if S_1 and S_2 are linearly equivalent. An algorithm for solving the linear equivalence problem is presented in [2]. The inputs to the algorithm are S_1 and S_2, and the output is either a linear equivalence (A, B) in case S_1 and S_2 are linearly equivalent, or a message that such a linear equivalence does not exist. The algorithm is referred to as the linear equivalence algorithm (LE), and exploits the linearity of the mappings A and B. For an in depth description LE, refer to [2]. Below we give a brief description of a variant of LE where it is assumed that both given S-boxes map 0 to itself, i.e., $S_1(0) = S_2(0) = 0$. This variant of LE will be used as a building block for the cryptanalysis in this paper.

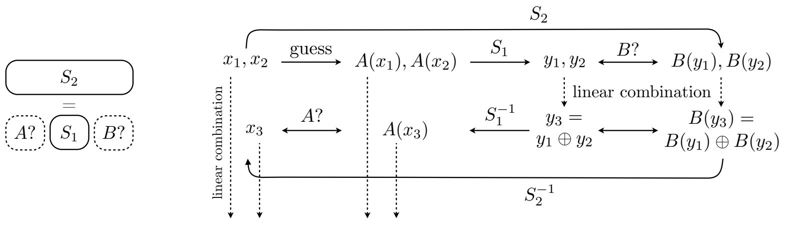


Fig. 3. Illustration how LE works

In case $S_1(0) = S_2(0) = 0$, at least two guesses for two points of A are necessary in order to start LE; select two distinct input points $x_1 \neq 0$ and $x_2 \neq 0$ and guess the values of $A(x_1)$ and $A(x_2)$. Based on these two initial guesses and the linearity of A and B, we incrementally build the linear mappings A and B as far as possible. The initial guesses $A(x_i)$ for the points x_i $(i = 1, 2)$ provide us with knowledge about B by computing $y_i = S_1(A(x_i))$ and $B(y_i) = S_2(x_i)$, which in turn gives us possibly new information about A by computing the images of the linear combinations of y_i and $B(y_i)$ through respectively S_1^{-1} and S_2^{-1}. This process is applied iteratively, where in each step of the process the linearity of the partially determined mappings A and B is verified by a Gaussian elimination. Figure 3 illustrates the process. In case neither for A nor for B a set of n linearly independent inputs and outputs is obtained, the algorithm requires an additional guess for a new point x of A (or B) in order to continue.

If n linearly independent inputs and n linearly independent outputs to A are obtained, then a candidate for A can be computed. Similar reasoning applies to B. If the candidate linear equivalence is denoted by (A^*, B^*), then the correctness of this pair can be tested by verifying the relation $S_2 = B^* \circ S_1 \circ A^*$ for all possible inputs. If no candidate linear equivalence is found (due to linear inconsistencies occurred during the process), or if the candidate linear equivalence is incorrect, then the process is repeated with a different guess for $A(x_1)$ or for $A(x_2)$, or for any of the possibly additional guesses made during the execution of LE.

The original linear equivalence algorithm LE exits after finding one single linear equivalence which already proves that both given S-boxes S_1 and S_2 are linearly equivalent. However, by running LE over all possible guesses, i.e., both initial guesses as well as the possibly additional guesses made during the execution of LE, also other linear equivalences (A, B) can be found. The work factor of this variant is at least $n^3 \cdot 2^{2n}$, i.e., a Gaussian elimination (n^3) for each possible pair of initial guesses (2^{2n}).

3 Cryptanalysis of the White-Box AES Implementation

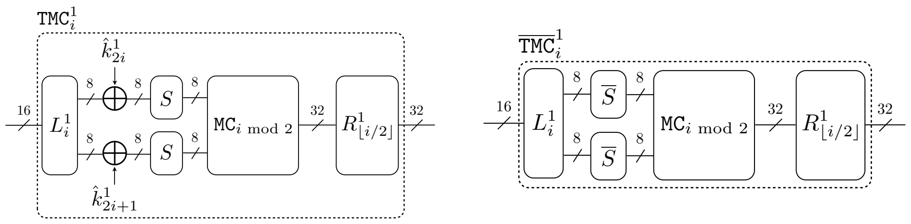
In this section, we elaborate on the cryptanalysis of the white-box AES-128 implementation proposed in [10] and described in Sect. 2.2. The goal of the cryptanalysis is the recovery of the full 128-bit AES key, together with the external input and output encodings, `IN` and `OUT` respectively.

The cryptanalysis focusses on extracting the first round key $\hat{k}^1$ contained within the 8 key-dependent 16-to-32 bit lookup tables $\mathtt{TMC}_i^1$ $(i = 0, \ldots, 7)$ of the first round. Each table $\mathtt{TMC}_i^1$, depicted in Fig. 4(a), is defined as follows:

$$\mathtt{TMC}_i^1 = R^1_{\lfloor i/2 \rfloor} \circ \mathtt{MC}_{i \bmod 2} \circ S \| S \circ \oplus_{(\hat{k}^1_{2i} \| \hat{k}^1_{2i+1})} \circ L_i^1 \quad , \tag{3}$$

where $\|$ denotes the concatenation symbol, $\oplus_c$ denotes the function $\oplus_c(x) = x \oplus c$, and $S\|S$ denotes the 16-bit bijective S-box comprising two AES S-boxes in parallel. Given (3), the adversary knows that both S-boxes $S_1 = S\|S$ and $S_2 = \mathtt{TMC}_i^1$ are affine equivalent by the affine equivalence $(A, B) = (\oplus_{(\hat{k}^1_{2i} \| \hat{k}^1_{2i+1})} \circ L_i^1, R^1_{\lfloor i/2 \rfloor} \circ \mathtt{MC}_{i \bmod 2})$ such that $S_2 = B \circ S_1 \circ A$. As one can notice, only A is affine where the constant part equals the key-material contained within $\mathtt{TMC}_i^1$. Hence by making $\mathtt{TMC}_i^1$ key-independent (see Lemma 1 below), we can reduce the problem to finding linear instead of affine equivalences, for which we apply the linear equivalence algorithm (LE).

Lemma 1. *Given the key-dependent 16-to-32 bit lookup table* $\mathtt{TMC}_i{}^1$ *(defined by* (3) *and depicted in Fig.* 4(a)*), let* x_0^i *be the 16-bit value such that* $\mathtt{TMC}_i{}^1(x_0^i) = 0$*, and let* $\overline{\mathtt{TMC}}_i{}^1$ *be defined as* $\overline{\mathtt{TMC}}_i{}^1 = \mathtt{TMC}_i{}^1 \circ \oplus_{x_0^i}$*. If* $\overline{S}$ *is defined as the 8-bit bijective S-box* $\overline{S} = S \circ \oplus_{\text{'52'}}$ *where* S *denotes the AES S-box, then:*



(a) Key-material ($\hat{k}_{2i}^1$ and $\hat{k}_{2i+1}^1$) (b) No key-material

Fig. 4. Key-dependent table $\texttt{TMC}_i^1$ versus key-independent table $\overline{\texttt{TMC}}_i^1$

$$\overline{TMC_i}^1 = R^1_{\lfloor i/2 \rfloor} \circ MC_{i \bmod 2} \circ \overline{S} \| \overline{S} \circ L_i^1 \ , \tag{4}$$

where $\overline{S}\|\overline{S}$ denotes the 16-bit bijective S-box comprising two S-boxes $\overline{S}$ in parallel. The key-independent 16-to-32 bit lookup table $\overline{TMC_i}^1$ is depicted in Fig. 4(b).

Proof. Given the fact that $\texttt{TMC}_i^1$ is encoded merely by linear input and output encodings L_i^1 and $R^1_{\lfloor i/2 \rfloor}$ (see (3)) and that $S(\text{‘}\texttt{52}\text{’}) = 0$, the 16-bit value x_0^i, for which $\texttt{TMC}_i{}^1(x_0^i) = 0$, has the following form:

$$x_0^i = (L_i^1)^{-1}\Big((\hat{k}_{2i}^1 \oplus \text{‘}\texttt{52}\text{’}) \parallel (\hat{k}_{2i+1}^1 \oplus \text{‘}\texttt{52}\text{’})\Big) \ , \tag{5}$$

In the rare case that $x_0^i = 0$, it immediately follows that both first round key bytes $\hat{k}_{2i}^1$ and $\hat{k}_{2i+1}^1$ are equal to ‘52’. Now, based on x_0^i, one can construct the key-independent 16-to-32 bit lookup table $\overline{\texttt{TMC}}_i^1$ as follows:

$$\begin{aligned}
\overline{\texttt{TMC}}_i^1 = \texttt{TMC}_i^1 \circ \oplus_{x_0^i} &= R^1_{\lfloor i/2 \rfloor} \circ \texttt{MC}_{i \bmod 2} \circ S \| S \circ \oplus_{(\hat{k}_{2i}^1 \| \hat{k}_{2i+1}^1)} \circ \oplus_{L_i^1(x_0^i)} \circ L_i^1 \\
&= R^1_{\lfloor i/2 \rfloor} \circ \texttt{MC}_{i \bmod 2} \circ S \| S \circ \oplus_{(\text{‘}\texttt{52}\text{’} \| \text{‘}\texttt{52}\text{’})} \circ L_i^1 \\
&= R^1_{\lfloor i/2 \rfloor} \circ \texttt{MC}_{i \bmod 2} \circ \overline{S} \| \overline{S} \circ L_i^1 \ . \qquad \square
\end{aligned}$$

The 8-bit bijective S-box $\overline{S}$ maps 0 to itself, i.e. $\overline{S}(\text{‘}\texttt{00}\text{’}) = \text{‘}\texttt{00}\text{’}$, since $S(\text{‘}\texttt{52}\text{’}) = \text{‘}\texttt{00}\text{’}$. Given (4), it also follows that $\overline{\texttt{TMC}}_i^1(0) = 0$.

Linear Equivalence Algorithm (LE). Given (4), the adversary knows that the 16-bit bijective S-box $S_1 = \overline{S}\|\overline{S}$ and the key-independent 16-to-32 bit lookup table $S_2 = \overline{\texttt{TMC}}_i^1$ (which is a bijective mapping from GF$\left(2^{16}\right)$ to a 16-dimensional subspace of GF$\left(2^{32}\right)$) obtained through Lemma 1, are linearly equivalent by the linear equivalence $(A, B) = (L_i^1, R^1_{\lfloor i/2 \rfloor} \circ \texttt{MC}_{i \bmod 2})$. His goal is to recover this linear equivalence which contains the secret linear input encoding L_i^1 he needs in order to extract both first round key bytes $\hat{k}_{2i}^1$ and $\hat{k}_{2i+1}^1$ out of the 16-bit value x_0^i given by (5). The described problem is exactly what the linear equivalence algorithm (LE) tries to solve.

Since in this case both S-boxes $S_1 = \overline{S}\|\overline{S}$ and $S_2 = \overline{\texttt{TMC}}_i^1$ map 0 to itself, at least two initial 16-bit guesses $A(x_n)$ for two distinct points $x_n \neq 0$ $(n = 1, 2)$ of A are necessary to execute LE, and hence the work factor becomes at least 2^{44}, i.e., $n^3 \cdot 2^{2n}$ for $n = 16$. Furthermore, 128 linear equivalences $(A, B) = (A^s \circ L_i^1, R_{\lfloor i/2 \rfloor}^1 \circ \texttt{MC}_{i \bmod 2} \circ B^s)$ can be found, where (A^s, B^s) denotes the 128 linear self-equivalences of $\overline{S}\|\overline{S}$ (see Appendix A):

$$\overline{S}\|\overline{S} = B^s \circ \overline{S}\|\overline{S} \circ A^s \quad \rightarrow$$
$$\overline{\texttt{TMC}}_i^1 = R_{\lfloor i/2 \rfloor}^1 \circ \texttt{MC}_{i \bmod 2} \circ B^s \circ \overline{S}\|\overline{S} \circ A^s \circ L_i^1 = B \circ \overline{S}\|\overline{S} \circ A \ .$$

The desired linear equivalence, i.e., the linear equivalence that the adversary wants to obtain, is denoted by $(A, B)_d = (L_i^1, R_{\lfloor i/2 \rfloor}^1 \circ \texttt{MC}_{i \bmod 2})$ and corresponds to the one with the linear self-equivalence $(A^s, B^s) = (I_{16}, I_{16})$, where I_{16} denotes the 16-bit identity matrix over GF (2).

Our Goal. In the following sections, we present a way how to modify the linear equivalence algorithm when applied to $S_1 = \overline{S}\|\overline{S}$ and $S_2 = \overline{\texttt{TMC}}_i^1$, such that only the single desired linear equivalence $(A, B)_d = (L_i^1, R_{\lfloor i/2 \rfloor}^1 \circ \texttt{MC}_{i \bmod 2})$ is given as output. At the same time, the work factor decreases as well. This modification exploits both the structure of AES as well as the structure of the white-box implementation.

3.1 Obtain Leaked Information about the Linear Input Encoding L_i^1

Due to the inherent structure of the white-box implementation, partial information about the linear input encoding L_i^1 of the key-independent tables $\overline{\texttt{TMC}}_i^1$ of the first round is leaked. For each L_i^1, this leaked information comprises four sets of 16-bit encoded values for which the underlying unencoded bytes share a known bijective function. In the next section, we show how to modify the linear equivalence algorithm based on this leaked information. Here, we elaborate on how this information is extracted for $L_{i^*}^1$ of a given table $\overline{\texttt{TMC}}_{i^*}^1$ for some $i^* \in \{0, 1, \ldots, 7\}$. Below, the following description is used: given an AES state by $[\text{state}_n]_{n=0,1,\ldots,15}$, then each set of 4 consecutive bytes $[\text{state}_{4j}, \text{state}_{4j+1}, \text{state}_{4j+2}, \text{state}_{4j+3}]$ for $j = 0, \ldots, 3$ is referred to as column j.

First, one builds an implementation which only consists of the single key-independent table $\overline{\texttt{TMC}}_{i^*}^1$ followed by the matrix multiplication over GF (2) with M^2 given by (1) for $r = 2$. This implementation is in detail depicted in Fig. 5 for $i^* = 4$, where the internal states U, V and Y are indicated as well: the 2-byte state $U = (u_0\|u_1)$ corresponds to the 2-byte input to $\overline{S}\|\overline{S}$ and the 4-byte state $V = (v_0\|v_1\|v_2\|v_3)$ corresponds to the 4-byte output of $\texttt{MC}_z$ where $z = i^* \bmod 2$. Since each byte v_l $(l = 0, \ldots, 3)$ of V is an output byte of $\texttt{MC}_z$, the relation between U and V is given by $mc_{l,0}^z \otimes \overline{S}(u_0) \oplus mc_{l,1}^z \otimes \overline{S}(u_1) = v_l$ for $l = 0, \ldots, 3$, where $\otimes$ denotes the multiplication over the Rijndael finite field GF $\left(2^8\right)$ and

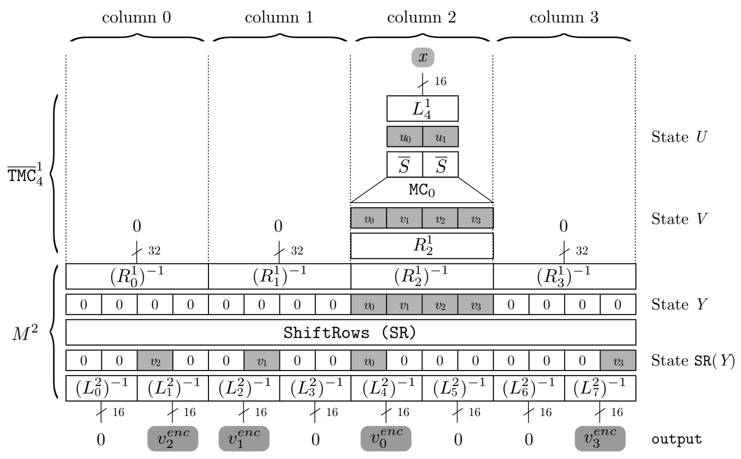

column 0
column 1
column 2
column 3
x
16
L_4^1
u_0 u_1
$\overline{S}$ $\overline{S}$
$\mathtt{MC}_0$
State U
State V
$\overline{\mathtt{TMC}}_4^1$
v_0 v_1 v_2 v_3
R_2^1
0
32
$(R_0^1)^{-1}$
$(R_1^1)^{-1}$
$(R_2^1)^{-1}$
$(R_3^1)^{-1}$
State Y
M^2
ShiftRows (SR)
State SR(Y)
$(L_0^2)^{-1}$ $(L_1^2)^{-1}$ $(L_2^2)^{-1}$ $(L_3^2)^{-1}$ $(L_4^2)^{-1}$ $(L_5^2)^{-1}$ $(L_6^2)^{-1}$ $(L_7^2)^{-1}$
v_2^{enc} v_1^{enc} v_0^{enc} v_3^{enc}
output

Fig. 5. Example of the implementation associated to $\overline{\mathtt{TMC}}_{i^*}^1$ with $i^* = 4$: identifying the four values v_l^{enc} for $l = 0, \ldots, 3$ in order to build the corresponding sets $\mathcal{S}_l^{i^*}$

the pair $(mc_{l,0}^z, mc_{l,1}^z)$ corresponds to the `MixColumns` coefficients on row l of the 4×2 submatrix $\mathtt{MC}_z$ over GF (2^8), i.e. $(mc_{l,0}^z, mc_{l,1}^z) \in \mathcal{S}_{\mathtt{MC}}$ with:

$$\mathcal{S}_{\mathtt{MC}} = \{(\text{`}\mathtt{02}\text{'}, \text{`}\mathtt{03}\text{'}), (\text{`}\mathtt{01}\text{'}, \text{`}\mathtt{02}\text{'}), (\text{`}\mathtt{01}\text{'}, \text{`}\mathtt{01}\text{'}), (\text{`}\mathtt{03}\text{'}, \text{`}\mathtt{01}\text{'})\} \ . \tag{6}$$

Then, the 16-byte input to M^2 is provided as follows: three 4-byte 0-values for columns $j \neq \lfloor i^*/2 \rfloor$ (in our example: $j = 0, 1, 3$) and the 4-byte output of $\overline{\mathtt{TMC}}_{i^*}^1$ for column $j = \lfloor i^*/2 \rfloor$ (in our example: $j = 2$). This ensures that the three columns j with $j \neq \lfloor i^*/2 \rfloor$ of the state Y remain zero, whereas column $j = \lfloor i^*/2 \rfloor$ equals the 4-byte state V. The `ShiftRows` operation ensures that the four bytes v_l ($l = 0, \ldots, 3$) of V are spread over all four columns of the internal state $\mathtt{SR}(Y)$, which are then each encoded by a different linear encoding $(L^2_{2((\lfloor i^*/2 \rfloor - l) \bmod 4) + \lfloor l/2 \rfloor})^{-1}$. Hence the `output` state contains four 16-bit 0-values, whereas the other four 16-bit output values v_l^{enc} ($l = 0, \ldots, 3$) each correspond to one of the 4 bytes v_l of V in a linearly encoded form.

If $v_l^{enc} = 0$, then the associated byte $v_l = \text{`}\mathtt{00}\text{'}$ as well, such that we have a known bijective function $f_l^{i^*}$ between the bytes u_0, u_1 of U, which is defined as:

$$u_1 = f_l^{i^*}(u_0) \text{ with } f_l^{i^*} = \left(\overline{S}\right)^{-1} \circ \otimes_{(mc_{l,1}^z)^{-1}} \circ \otimes_{mc_{l,0}^z} \circ \overline{S} \ , \tag{7}$$

where $z = i^* \bmod 2$. This function follows out of the equation $mc_{l,0}^z \otimes \overline{S}(u_0) \oplus mc_{l,1}^z \otimes \overline{S}(u_1) = \text{`}\mathtt{00}\text{'}$ and is depicted in Fig. 6.

Now, for the linear input encoding $L_{i^*}^1$ of $\overline{\mathtt{TMC}}_{i^*}^1$, four sets $\mathcal{S}_l^{i^*}$ ($l = 0, \ldots, 3$) are built as follows. First associate one of the four values v_l^{enc} with each set $\mathcal{S}_l^{i^*}$. Then, for each set $\mathcal{S}_l^{i^*}$, store the 16-bit value x, given as input to $\overline{\mathtt{TMC}}_{i^*}^1$, for which the associated output value $v_l^{enc} = 0$. Do this for all $x \in$ GF (2^{16}). This results

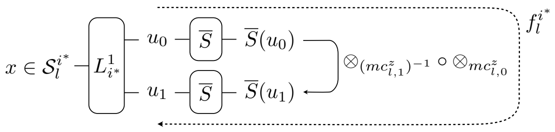


Fig. 6. How the known bijective function $f_l^{i^*}$ between u_0 and u_1 is defined

in that each set $\mathcal{S}_l^{i^*}$ is composed of 2^8 16-bit encoded values x for which the underlying unencoded bytes u_0, u_1 share the known bijective function $f_l^{i^*}$ given by (7) and depicted in Fig. 6:

$$\mathcal{S}_l^{i^*} = \{x \in \mathrm{GF}\left(2^{16}\right) \mid L_{i^*}^1(x) = u_0 \| u_1 \wedge u_1 = f_l^{i^*}(u_0)\} \quad \text{with } |\mathcal{S}_l^{i^*}| = 2^8 \ . \quad (8)$$

So with each set $\mathcal{S}_l^{i^*}$ $(l = 0, \ldots, 3)$, a known bijective function $f_l^{i^*}$ is associated.

3.2 Finding the Desired Linear Equivalence $(A, B)_d$: Obtain the Full Linear Input Encoding L_i^1

So far, for the secret linear input encoding L_i^1 of each table $\overline{\mathtt{TMC}}_i^1$ $(i = 0, 1, \ldots, 7)$ of the first round, four sets $\mathcal{S}_l^i$ $(l = 0, \ldots, 3)$ defined by (8) are obtained. For each element $x \in \mathcal{S}_l^i$, the underlying unencoded bytes u_0, u_1 share a specific known bijective function f_l^i given by (7) and depicted in Fig. 6. Now, by exploiting this leaked information about L_i^1, we present an efficient algorithm for computing the desired linear equivalence $(A, B)_d = (L_i^1, R_{\lfloor i/2 \rfloor}^1 \circ \mathtt{MC}_{i \bmod 2})$. This enables the adversary to obtain the secret linear input encoding $A = L_i^1$ of $\overline{\mathtt{TMC}}_i^1$, which also corresponds to the linear input encoding of $\mathtt{TMC}_i^1$.

Algorithm for Finding $(A, B)_d$. Since $A = L_i^1$ in the desired linear equivalence, we exploit the leaked information about L_i^1 in order to make the two initial guesses $A(x_n)$ for two distinct points $x_n \neq 0$ $(n = 1, 2)$ of A. Only two out of four sets $\mathcal{S}_l^i$ are considered, i.e., those where the pair of `MixColumns` coefficients $(mc_{l,0}^z, mc_{l,1}^z)$ of the associated function f_l^i equals (‘`01`’, ‘`02`’) or (‘`02`’, ‘`03`’). We choose one of both sets and simply denote it by $\mathcal{S}^i$.

Now, select two distinct points $x_n \neq 0$ $(n = 1, 2)$ out of the chosen set, i.e., $x_n \in \mathcal{S}^i$. Based on definition (8) of $\mathcal{S}^i$, these points are defined as $x_n = (L_i^1)^{-1}\big(u_n \| f^i(u_n)\big)$ for some unknown distinct 8-bit values $u_n \in \mathrm{GF}\left(2^8\right) \setminus \{0\}$, where f^i denotes the known function associated with $\mathcal{S}^i$. Now, based on this knowledge and the fact that we want to find $A = L_i^1$, the two initial guesses $A(x_n)$ are made as follows: $A(x_n) = a_n \| f^i(a_n)$ for all $a_n \in \mathrm{GF}\left(2^8\right) \setminus \{0\}$ $(n = 1, 2)$. Hence although $A(x_n)$ is a 16-bit value, we only need to guess the 8-bit value a_n such that the total number of guesses becomes 2^{16} (i.e. $2^{2\frac{n}{2}}$ with $n = 16$). For each possible pair of initial guesses $\big(A(x_n) = a_n \| f^i(a_n)\big)_{n=1,2}$, LE is executed on $S_1 = \overline{S} \| \overline{S}$ and $S_2 = \overline{\mathtt{TMC}}_i^1$. All found linear equivalences are stored in the set $\mathcal{S}_{\mathrm{LE}}$.

It is assumed that at least $(A,B)_d = (L_i^1, R^1_{\lfloor i/2 \rfloor} \circ \mathtt{MC}_{i \bmod 2}) \in \mathcal{S}_{\mathrm{LE}}$, which occurs when $a_n = u_n$ for $n = 1,2$. It is possible that one or more linear equivalences $(A,B) = (A^s \circ L_i^1, R^1_{\lfloor i/2 \rfloor} \circ \mathtt{MC}_{i \bmod 2} \circ B^s)$ with $A^s \neq I_{16}$ (see the introducing part of Sect. 3) can be found as well such that $|\mathcal{S}_{\mathrm{LE}}| > 1$. In that case, the procedure needs to be repeated for two new distinct points $x_n^* \neq 0$ $(n = 1,2)$ out of the chosen set $\mathcal{S}^i$, which are also distinct from the original chosen points $x_n \neq 0$ $(n = 1,2)$. This results in a second set $\mathcal{S}^*_{\mathrm{LE}}$. Assuming that all possible linear equivalences between $S_1 = \overline{S} \| \overline{S}$ and $S_2 = \overline{\mathtt{TMC}}_i^1$ are given by $(A,B) = (A^s \circ L_i^1, R^1_{\lfloor i/2 \rfloor} \circ \mathtt{MC}_{i \bmod 2} \circ B^s)$, it can be shown that for both considered sets, it is impossible that a linear equivalence with $A^s \neq I_{16}$ is given as output during both executions of the procedure. Hence taking the intersection of both sets $\mathcal{S}_{\mathrm{LE}}$ and $\mathcal{S}^*_{\mathrm{LE}}$ results in the desired linear equivalence $(A,B)_d$.

Algorithm 1 gives a detailed description of the whole procedure. It has an average case work factor of 2^{29}, i.e., $2 \cdot n^3 \cdot 2^{2\frac{n}{2}}$ for $n = 16$.

Algorithm 1. Finding the desired linear equivalence $(A,B)_d$

```
Input: $S_1 = \overline{S} \| \overline{S}$, $S_2 = \overline{\mathtt{TMC}}_i^1$, $\mathcal{S}^i$, $f^i$
Output: $(A,B)_d = (L_i^1, R^1_{\lfloor i/2 \rfloor} \circ \mathtt{MC}_{i \bmod 2})$
1: select two distinct points $x_1, x_2 \in \mathcal{S}^i$ with $x_n \neq 0$ $(n = 1,2)$
2: call search-LE$(x_1, x_2) \to \mathcal{S}_{\mathrm{LE}}$
3: if $|\mathcal{S}_{\mathrm{LE}}| > 1$ then
4:     select two distinct points $x_1^*, x_2^* \in \mathcal{S}^i$ with $x_n^* \neq 0$, $x_n^* \neq x_m$ $(n = 1,2\ ;\ m = 1,2)$
5:     call search-LE$(x_1^*, x_2^*) \to \mathcal{S}^*_{\mathrm{LE}}$
6:     $\mathcal{S}_{\mathrm{LE}} \leftarrow \mathcal{S}_{\mathrm{LE}} \cap \mathcal{S}^*_{\mathrm{LE}}$
7: end if
8: return $\mathcal{S}_{\mathrm{LE}}$
where
Procedure search-LE (Input: $x_1, x_2$ – Output: $\mathcal{S}_{\mathrm{LE}}$)
1: $\mathcal{S}_{\mathrm{LE}} \leftarrow \varnothing$
2: for all $a_1 \in \mathrm{GF}\left(2^8\right) \setminus \{0\}$ do
3:     $A(x_1) \leftarrow a_1 \| f^i(a_1)$
4:     for all $a_2 \in \mathrm{GF}\left(2^8\right) \setminus \{0\}$ do
5:         $A(x_2) \leftarrow a_2 \| f^i(a_2)$
6:         call LE on $S_1 = \overline{S} \| \overline{S}$ and $S_2 = \overline{\mathtt{TMC}}_i^1$ with initial guesses $A(x_1), A(x_2) \to \mathcal{S}_{\mathrm{LE}}$
7:     end for
8: end for
```

Choice of Set $\mathcal{S}^i$. To each of the four sets $\mathcal{S}_l^i$ $(l = 0, \dots, 3)$, a pair of `MixColumns` coefficients $(mc^z_{l,0}, mc^z_{l,1}) \in \mathcal{S}_{\mathtt{MC}}$ (see (6)) of the associated function f_l^i is related. Let us denote this relation by $\mathcal{S}_l^i \leftrightarrow (mc^z_{l,0}, mc^z_{l,1})$. Here, we elaborate on the fact that not all four sets are equally suitable to be used in Algorithm 1.

$\mathcal{S}_l^i \leftrightarrow$ (‘01’, ‘01’): in this case, the associated function f_l^i becomes the identity function such that the pair of initial guesses becomes $\left(A(x_n) = a_n \| a_n\right)_{n=1,2}$ with $a_n \in \mathrm{GF}\left(2^8\right) \setminus \{0\}$. When executing LE on $S_1 = \overline{S} \| \overline{S}$ and $S_2 = \overline{\mathtt{TMC}}_i^1$ for any such pair, we only find at most 8 linearly independent inputs and output to A (or B). This is explained by the fact that linear combinations of $a_n \| a_n$ (or of $\overline{S}(a_n) \| \overline{S}(a_n)$) span at most an 8-dimensional space. In order to continue executing LE, an additional guess for a new point x of A (or B) is required which increases the work factor. Hence we avoid using this set;

$\mathcal{S}_l^i \leftrightarrow$ {(‘01’, ‘02’), (‘02’, ‘03’), (‘03’, ‘01’)}: computer simulations show that all three remaining sets can be used in Algorithm 1 without requiring an additional guess during the execution of LE. However, in the worst case scenario, using the set $\mathcal{S}_l^i \leftrightarrow$ (‘03’, ‘01’) requires that the procedure ‘`search-LE`’ needs to be executed 4 times in total in order to find the single desired linear equivalence $(A, B)_d$, instead of at most 2 times in case of the set $\mathcal{S}_l^i \leftrightarrow$ (‘01’, ‘02’) or the set $\mathcal{S}_l^i \leftrightarrow$ (‘02’, ‘03’). Note that Algorithm 1 assumes that one of the latter sets is chosen.

Implementation. Algorithm 1 has been implemented in `C++` and tests have been conducted on an Intel Core2 Quad @ 3.00GHZ. For the conducted tests, we chose the set $\mathcal{S}_l^i$ where $(mc_{l,0}^z, mc_{l,1}^z) =$ (‘02’, ‘03’). We ran the implementation 3000 times in total, each time for a different randomly chosen L_i^1 and $R_{\lfloor i/2 \rfloor}^1$. Only 4 times the procedure ‘`search-LE`’ needed to be repeated since 2 linear equivalences were found during the first execution. The implementation always succeeded in finding only the single desired linear equivalence $(A, B)_d$, which required on average $\approx$ 1min. It should be noted that the implementation was not optimized for speed, hence improvements are possible. The implementation also showed that each pair of initial guesses as defined above were sufficient in order to execute LE, i.e., no additional guesses were required.

3.3 Extracting the Full 128-Bit AES Key and the External Input and Output Encodings `IN` and `OUT`

At this point in the cryptanalysis, we extracted the 16-bit secret linear input encodings L_i^1 of all 8 16-to-32 bit tables $\mathtt{TMC}_i^1$ $(i = 0, \ldots, 7)$ of the first round.

Extracting the Full 128-bit AES Key. Given the 16-bit value x_0^i of each table $\mathtt{TMC}_i^1$ defined by $\mathtt{TMC}_i^1(x_0^i) = 0$ (see (5)), the adversary can extract both first round key bytes $\hat{k}_{2i}^1$ and $\hat{k}_{2i+1}^1$ contained within each key-dependent table $\mathtt{TMC}_i^1$ as follows:

$$\hat{k}_{2i}^1 \parallel \hat{k}_{2i+1}^1 = L_i^1(x_0^i) \oplus \left(\text{‘52’} \parallel \text{‘52’}\right) .$$

By doing so for each table $\mathtt{TMC}_i^1$ $(i = 0, 1, \ldots, 7)$ and taking into account the data flow of the white-box implementation of the first round, the adversary is able to obtain the full 128-bit first round key $\hat{k}^1$, which after applying the inverse `ShiftRows` operation to it results in the actual first round key k^1. According to the AES key scheduling algorithm, k^1 corresponds to the 128-bit AES key k.

Extracting the External Input and Output Encodings `IN` *and* `OUT`*.* By knowing all 8 linear input encodings L_i^1 ($i = 0, 1, \ldots, 7$) of the first round, the external 128-bit linear input encoding `IN` can be extracted out of the 128×128 binary matrix M^1 given by (2) as follows: $\mathtt{IN}^{-1} = \mathtt{SR}^{-1} \circ \mathrm{diag}(L_0^1, \ldots, L_7^1) \circ M^1$.

The external 128-bit linear output encoding `OUT` can be extracted once both the AES key k and `IN` have been recovered. Let us take the canonical base $([e_i])_{i=0,\ldots,127}$ of the vector space $\mathrm{GF}(2)^{128}$ and calculate for each 128-bit base vector e_i the 128-bit value $y_i = \mathrm{WBAES}_k\big(\mathtt{IN}\big(\mathrm{AES}_k^{-1}(e_i)\big)\big)$, where WBAES_k denotes the given white-box AES implementation defined by $\mathrm{WBAES}_k = \mathtt{OUT} \circ \mathrm{AES}_k \circ \mathtt{IN}^{-1}$ and AES_k^{-1} denotes the inverse standard AES implementation, both instantiated with the AES key k:

$$y_i = \underbrace{\mathtt{OUT}(\mathrm{AES}_k(\mathtt{IN}^{-1}}_{\mathrm{WBAES}_k}(\mathtt{IN}(\mathrm{AES}_k^{-1}(e_i))))) = \mathtt{OUT}(e_i) \ .$$

As one can notice, y_i corresponds to the image of e_i under the external 128-bit linear output encoding `OUT`. Hence `OUT` is completely defined by calculating all pairs (e_i, y_i) for $i = 0, \ldots, 127$.

3.4 Work Factor

The overall work factor of our cryptanalysis is dominated by the execution of Algorithm 1 in order to obtain the linear input encodings L_i^1 of all 8 16-to-32 bit tables $\mathtt{TMC}_i^1$ ($i = 0, \ldots, 7$) of the first round. The algorithm has a work factor of about 2^{29}. Thus, executing the algorithm on $S_1 = \overline{S} \| \overline{S}$ and $S_2 = \overline{\mathtt{TMC}_i^1}$ for $i = 0, 1, \ldots, 7$ leads to an overall work factor of about $8 \cdot 2^{29} = 2^{32}$. Once obtained L_i^1 for $i = 0, 1, \ldots, 7$, the AES key together with the external encodings can be extracted as explained in Sect. 3.3.

4 Conclusion

This paper described in detail a practical attack on the white-box AES implementation of Xiao and Lai [10]. The cryptanalysis exploits both the structure of AES as well as the structure of the white-box AES implementation. It uses a modified variant of the linear equivalence algorithm presented by Biryukov *et al.* [2], which is built by exploiting leaked information out of the white-box implementation. The attack efficiently extracts the AES key from Xiao *et al.*'s white-box AES implementation with a work factor of about 2^{32}. In addition to extracting the AES key, which is the main goal in cryptanalysis of white-box implementations, our cryptanalysis is also able to recover the external input and output encodings. Crucial parts of the cryptanalysis have been implemented in `C++` and verified by computer experiments. The implementation furthermore shows that both the 128-bit AES key as well as the external input and output encodings can be extracted from the white-box implementation in just a few minutes on a modern PC.

Acknowledgements. This work was supported in part by the Research Council KU Leuven: GOA TENSE (GOA/11/007), by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy) and by the European Commission through the ICT Programme under contract ICT-2007-216676 ECRYPT II. In addition, this work was supported by the Flemish Government, FWO WET G.0213.11N and IWT GBO SEC SODA. Yoni De Mulder was supported in part by IBBT (Interdisciplinary institute for BroadBand Technology) of the Flemish Government.

References

1. Billet, O., Gilbert, H., Ech-Chatbi, C.: Cryptanalysis of a White Box AES Implementation. In: Handschuh, H., Hasan, M.A. (eds.) SAC 2004. LNCS, vol. 3357, pp. 227–240. Springer, Heidelberg (2004)
2. Biryukov, A., De Cannière, C., Braeken, A., Preneel, B.: A Toolbox for Cryptanalysis: Linear and Affine Equivalence Algorithms. In: Biham, E. (ed.) EUROCRYPT 2003. LNCS, vol. 2656, pp. 33–50. Springer, Heidelberg (2003)
3. Bringer, J., Chabanne, H., Dottax, E.: White box cryptography: Another attempt. Cryptology ePrint Archive, Report 2006/468 (2006), http://eprint.iacr.org/2006/468.pdf
4. Chow, S., Eisen, P.A., Johnson, H., van Oorschot, P.C.: White-Box Cryptography and an AES Implementation. In: Nyberg, K., Heys, H.M. (eds.) SAC 2002. LNCS, vol. 2595, pp. 250–270. Springer, Heidelberg (2003)
5. Chow, S., Eisen, P., Johnson, H., van Oorschot, P.C.: A White-Box DES Implementation for DRM Applications. In: Feigenbaum, J. (ed.) DRM 2002. LNCS, vol. 2696, pp. 1–15. Springer, Heidelberg (2003)
6. Karroumi, M.: Protecting White-Box AES with Dual Ciphers. In: Rhee, K.-H., Nyang, D. (eds.) ICISC 2010. LNCS, vol. 6829, pp. 278–291. Springer, Heidelberg (2011)
7. Muir, J.A.: A tutorial on white-box AES. Mathematics in Industry (2012), http://www.ccsl.carleton.ca/~jamuir/papers/wb-aes-tutorial.pdf
8. De Mulder, Y., Wyseur, B., Preneel, B.: Cryptanalysis of a Perturbated White-Box AES Implementation. In: Gong, G., Gupta, K.C. (eds.) INDOCRYPT 2010. LNCS, vol. 6498, pp. 292–310. Springer, Heidelberg (2010)
9. National Institute of Standards and Technology. Advanced encryption standard. FIPS publication 197 (2001), http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf
10. Xiao, Y., Lai, X.: A secure implementation of white-box AES. In: 2nd International Conference on Computer Science and its Applications (CSA 2009), pp. 1–6. IEEE (2009)

A Linear Self-equivalences of 16-bit Bijective S-box $\overline{S}||\overline{S}$

Let the AES S-box S be defined as $S = A(x^{-1})$ where A is a 8-bit bijective affine mapping over GF (2) and x^{-1} denotes the inverse of x in the Rijndael finite field GF $\left(2^8\right)$ with '00' $^{-1}$ = '00', and let the 8-bit bijective S-box $\overline{S}$ be defined as

$\overline{S} = S \circ \oplus_{\text{‘52’}}$. If $\varPhi_l$ denotes the set of exactly $\#_l = 8$ linear self-equivalences (α, β) of $\overline{S}$ such that $\overline{S} = \beta \circ \overline{S} \circ \alpha$, and is defined as:

$$\varPhi_l = \left\{ (\alpha = [c] \circ Q^i, \beta = A \circ Q^{-i} \circ [c] \circ A^{-1}) \mid (i, c) \in \mathcal{S}_l \right\} \qquad \text{with}$$
$$\mathcal{S}_l = \{(0, \text{‘}\texttt{01}\text{’}), (1, \text{‘}\texttt{05}\text{’}), (2, \text{‘}\texttt{13}\text{’}), (3, \text{‘}\texttt{60}\text{’}), (4, \text{‘}\texttt{55}\text{’}), (5, \text{‘}\texttt{f6}\text{’}), (6, \text{‘}\texttt{b2}\text{’}), (7, \text{‘}\texttt{66}\text{’})\} \ ,$$

where $[c]$ denotes the 8×8 binary matrix representing a multiplication by c in $\mathrm{GF}\left(2^8\right)$ and Q denotes the 8×8 binary matrix that performs the squaring operation in $\mathrm{GF}\left(2^8\right)$ (both considered the Rijndael finite field), then the 16-bit bijective S-box comprising two identical S-boxes $\overline{S}$ in parallel, i.e. $\overline{S} \| \overline{S}$, has $2 \cdot \#_l^2 = 128$ trivial linear self-equivalences denoted by the pair of 16-bit bijective linear mappings (A^s, B^s) such that $\overline{S} \| \overline{S} = B^s \circ \overline{S} \| \overline{S} \circ A^s$, with the following diagonal structure:

$$A^s = \begin{pmatrix} \alpha_1 & 0_{8\times 8} \\ 0_{8\times 8} & \alpha_2 \end{pmatrix} \ , B^s = \begin{pmatrix} \beta_1 & 0_{8\times 8} \\ 0_{8\times 8} & \beta_2 \end{pmatrix} \text{ or}$$
$$A^s = \begin{pmatrix} 0_{8\times 8} & \alpha_1 \\ \alpha_2 & 0_{8\times 8} \end{pmatrix} \ , B^s = \begin{pmatrix} 0_{8\times 8} & \beta_2 \\ \beta_1 & 0_{8\times 8} \end{pmatrix} \ , \qquad (9)$$

for any combination $[(\alpha_1, \beta_1), (\alpha_2, \beta_2)]$ where both $(\alpha_1, \beta_1), (\alpha_2, \beta_2) \in \varPhi_l$. In (9), $0_{8\times 8}$ denotes the 8×8 binary zero-matrix over $\mathrm{GF}(2)$.

The linear equivalence algorithm (implemented in `C++`) has been executed with $S_1 = S_2 = \overline{S} \| \overline{S}$ and found exactly these 128 linear self-equivalences (A^s, B^s) of the form (9).

A Practical Leakage-Resilient Signature Scheme in the Generic Group Model*

David Galindo and Srinivas Vivek

University of Luxembourg
{david.galindo,srinivasvivek.venkatesh}@uni.lu

Abstract. We propose a leakage-resilient signature scheme in the continual leakage model that is based on a well-known identity-based encryption scheme by Boneh and Boyen (Eurocrypt 2004). The proposed signature scheme is the most efficient among the existing schemes that allow for continual leakage. Its efficiency is close to that of non leakage-resilient pairing-based signature schemes. It tolerates leakage of almost half of the bits of the secret key at every new signature invocation. We prove the security of the new scheme in the generic bilinear group model.

Keywords: leakage-resilient cryptography, digital signature, continual leakage, generic group model, efficiency.

1 Introduction

Side channel attacks are often effective in recovering the secret key of cryptosystems that are provably secure otherwise [16,17,7]. Typical examples of side channel attacks include analysis of running-time, power consumption, electromagnetic radiation leak, fault detection, to name just but a few. Countermeasures adopted in practice against side channel attacks are usually heuristic, aimed often at covering a restricted class of attacks. On the other hand, it is desirable to extend the traditional provable security methodology to also include side channel attacks. This area of contemporary cryptography is usually referred to as *leakage-resilient cryptography* and it has been an increasingly active area in recent years.

In this work we make two main assumptions to model leakage:

– **Bounded leakage:** the useful leakage data per signature invocation is bounded in length (but unbounded overall);
– **Independent leakage:** the computation can be divided into rounds, where each such round leaks independently.

This model has been previously used in [11,21,15,12]. The first assumption can be seen overly restrictive; however it should be noticed that in practice many side-channel attacks only exploit a polylogarithmic amount of information. The

* Supported by the National Research Fund, Luxembourg C09/IS/04.

second assumption allows us to divide the memory of a device, at every computing step, into two parts - an *active* and a *passive* part. The part of the memory being currently accessed by a computation is the active part, and only the active part leaks information at any given time. We stress that even if our leakage definition is local with respect to each part of the memory, it still captures some global functions of the secret key, for instance any affine leakage function. We refer to the work by Dziembowski and Faust [10] for a discussion on the significance and limitations of this leakage model. In particular, the Only Computation Leaks Information model [13,20] complies with our leakage model.

In the last few years a tremendous progress has been made in the interplay between provable security and side-channel attacks, such as the works [14,12,6,8,18,5] bear witness for the case of digital signatures. Admittedly, the schemes that do not use any idealized assumption (random oracle, generic groups), are much more involved than their non-leakage counterparts, and more importantly, not yet quite efficient to be used in practice. A rough estimation of the efficiency of current leakage-resilient schemes is that they are a linear number of times in the security parameter slower than their non-leakage counterparts. In this work we aim at building an efficient signature scheme secure against continual leakage. To this aim, we use an idealized model of computation called *generic bilinear group* (GBG) model, which has been previously used by Kiltz and Pietrzak [15] to provide leakage-resilient public key encryption. They propose a bilinear version of the ElGamal key encapsulation mechanism which enjoys provable leakage-resilience in the presence of continual leakage. Their scheme is very efficient, less than a handful of times slower than standard ElGamal.

We use the techniques by Kiltz and Pietrzak to propose a leakage-resilient signature scheme that builds upon the Boneh-Boyen identity-based encryption scheme [2]. The resulting signature scheme is nearly as efficient as the original identity-based encryption scheme (only $\frac{4}{3}$ times slower). Our main theorem (Theorem 2) states that allowing λ bits of leakage at every round decreases the security of the scheme by at most a factor $2^{2\lambda}$.

The main criticism that can be addressed to our work is the use of the generic group idealization to reason about side-channel attacks. The main question is whether the generic group model is a risky abstraction when side-channel attacks are considered. The main advantage of our chosen approach lies on its practicality: the schemes obtained are of efficiency comparable to traditional schemes, a major argument in our opinion to motivate the cryptographic engineering community's interest. This alone justifies in our view a careful consideration of this approach to reason about leakage-resilient schemes, since this level of practicality is still out of reach for the existing leakage-resilient schemes in the standard model. We would like to mention that nevertheless, given the breakthroughs achieved in the last few years in the theory of leakage-resilient cryptography, we are confident that the above-mentioned efficiency gap will be progressively shrunk in the years to come and under widely accepted assumptions.

2 Definitions

In this section, we recollect some basic notions of security of signature schemes, bilinear groups, and the generic bilinear group model. We also describe the model of leakage we shall consider in this paper and formulate a definition of security of signature schemes in the presence of continual leakage. We adapt the leakage model specified in [15] to signature schemes.

Let $\mathbb{Z}$ denote the set of integers and $\mathbb{Z}_p$ $(p > 0)$ denote, depending upon the context, either the set of integers $\{0, 1, \ldots, p-1\}$ or the ring modulo p. We denote a random sampling of an element $a \in A$ from a set A, and also denote a (possibly probabilistic) output of an algorithm A, by $a \leftarrow A$. If we want to explicitly denote the randomness r used during the sampling/output, then we do so by $s \overset{r}{\leftarrow} S$. Unless otherwise mentioned or implicit from the context, any sampling is from an uniform distribution. The symbol ":=" is used to define a notation in an expression, as in $A := \mathbb{Z}$, or to explicitly indicate an output of a deterministic algorithm or a function.

2.1 Existential Unforgeability

A signature scheme $\Pi = (\mathsf{KeyGen}, \mathsf{Sign}, \mathsf{Verify})$ consists of three probabilistic polynomial-time algorithms KeyGen, Sign, and Verify. Let κ denote the security parameter. $\mathsf{KeyGen}(\kappa)$ on input κ produces a public- and secret-key pair (pk, sk) along with other public parameters $\mathbb{PP}$. The algorithm $\mathsf{Sign}(sk, m)$ on input a secret key sk and a message $m \in M$, where M is the message space, outputs a signature σ. $\mathsf{Verify}(pk, m, \sigma)$ on input a public key pk, a message $m \in M$ and a signature σ, outputs a bit $b = 1$ meaning *valid*, or $b = 0$ meaning *invalid.* We require the following *correctness* requirement to be satisfied by Π:

$$\Pr[\mathsf{Verify}(pk, m, \mathsf{Sign}(sk, m)) = 1 \ : \ (pk, sk) \leftarrow \mathsf{KeyGen}(\kappa), m \in M] = 1.$$

The security of a signature scheme Π is defined through the following experiment:

```
Sign-Forge_Π(A, κ)                                  | Sign-Oracle $\Omega_{sk}(m)$
    $(pk, sk) \leftarrow \mathsf{KeyGen}(\kappa)$              |     $w := w \cup m$
    $w := \emptyset$                                        |     $\sigma \leftarrow \mathsf{Sign}(sk, m)$
    $(m, \sigma) \leftarrow \mathcal{A}^{\Omega_{sk}(\cdot)}(pk)$              |     Return $\sigma$
    If $m \in w$, then return $b := 0$                |
    $b \leftarrow \mathsf{Verify}(pk, m, \sigma)$                 |
```

Definition 1. [Existential Unforgeability] *A signature scheme* Π *is* existentially unforgeable under adaptive chosen-message attacks*, in short "secure", if* $\Pr[b = 1]$ *is negligible in the Experiment* Sign-Forge$_\Pi(\mathcal{A}, \kappa)$ *for any efficient adversary* $\mathcal{A}$*.*

2.2 Leakage Model

We split the secret state into two parts that reside in different parts of the memory, and structure any computation that involves access to the secret state into a sequence of steps. Any step accesses only one part of the secret state (*active* part) and the other part (*passive* part) is assumed not to leak in the current step of computation. In the case of signature schemes, we structure the signing process into two steps. For simplicity, we define a security notion for leakage-resilient signature schemes assuming that the signing process is carried out in two steps. We also refer to a single invocation of the signature generation algorithm as a *round*.

Let us consider the problem of achieving leakage resilience under continual leakage even when a significant fraction of the bits of the secret state are leaked per round. Then it is necessary that the secret state must be *stateful*, i.e. the secret state must be refreshed during every round [15]. Otherwise, after many rounds the entire secret state will be completely leaked.

Formally, a stateful signature scheme $\mathsf{\Pi}^* = (\mathsf{KeyGen}^*, \mathsf{Sign}_1^*, \mathsf{Sign}_2^*, \mathsf{Verify}^*)$ consists of four probabilistic polynomial-time algorithms KeyGen^*, Sign_1^*, Sign_2^* and Verify^*. $\mathsf{KeyGen}^*(\kappa)$ is same as the set-up phase KeyGen of $\mathsf{\Pi}$ except that instead of a "single" secret key sk, it outputs two initial secret states (S_0, S_0'). Intuitively, S_0 and S_0' may be viewed as two shares of the secret key sk. From the point of view of an adversary, the signing algorithm Sign of $\mathsf{\Pi}$ and $(\mathsf{Sign}_1^*, \mathsf{Sign}_2^*)$ have the same functionality. First, Sign_1^* is executed and later Sign_2^* is executed. That is, the i^{th} execution of the signing process (or i^{th} round) is carried out as:

$$(S_i, w_i) \overset{r_i}{\leftarrow} \mathsf{Sign}_1^*(S_{i-1}, m_i) \ ; \ (S_i', \sigma_i) \overset{r_i'}{\leftarrow} \mathsf{Sign}_2^*(S_{i-1}', w_i). \tag{1}$$

In the above expression, r_i and r_i' are the randomness used by Sign_1^* and Sign_2^*, respectively. The parameter w_i is some state information passed onto Sign_2^* by Sign_1^*. The signature σ_i is generated for the message m_i, and the internal state is updated from (S_{i-1}, S_{i-1}') to (S_i, S_i').

We model the leakage during signature generation by giving an adversary $\mathcal{A}$ access to a leakage oracle $\Omega^{\text{leak}}_{(S_{i-1}, S_{i-1}')}(\cdot)$. This oracle, in addition to giving $\mathcal{A}$ signatures for the messages of its choice, also allows $\mathcal{A}$ to obtain leakage from the computation used to generate signatures. More precisely, let λ be a *leakage parameter*. During the i^{th} signing round, $\mathcal{A}$ is allowed to specify two functions f_i and h_i, each of range $\{0,1\}^\lambda$, that can be efficiently computed. The outputs of the leakage functions are

$$\Lambda_i = f_i(S_{i-1}, r_i) \ ; \ \Lambda_i' = h_i(S_{i-1}', r_i', w_i). \tag{2}$$

Since the value of m can be included in the description of f_i and h_i, hence it is not explicitly included as an input. Note that it also possible for $\mathcal{A}$ to specify h_i after obtaining Λ_i. But, for the simplicity of the exposition, we focus on the case where f_i and h_i are specified along with the message m_i to the oracle. The security of the signature scheme $\mathsf{\Pi}^*$ in the presence of (continual) leakage is defined through the following experiment Sign-Forge-Leak$_{\Pi^*}(\mathcal{A}, \kappa, \lambda)$. In the description below, $|f_i|$ refers to the length of the output of f_i.

```
Sign-Forge-Leak$_{\Pi^*}(\mathcal{A}, \kappa, \lambda)$
    $(pk, (S_0, S'_0)) \leftarrow \mathsf{KeyGen}^*(\kappa)$
    $i := 1, w := \emptyset$
    $(m, \sigma) \leftarrow \mathcal{A}^{\Omega^{\text{leak}}_{(S_{i-1}, S'_{i-1})}(\cdot)}(pk)$
    If $m \in w$, then return $b := 0$
    $b \leftarrow \mathsf{Verify}^*(pk, m, \sigma)$

Sign-Leak-Oracle $\Omega^{\text{leak}}_{(S_{i-1}, S'_{i-1})}(m_i, f_i, h_i)$
    If $|f_i| \neq \lambda$ or $|h_i| \neq \lambda$, return $\perp$
    $(S_i, w_i) \xleftarrow{r_i} \mathsf{Sign}_1^*(S_{i-1}, m_i)$
    $(S'_i, \sigma_i) \xleftarrow{r'_i} \mathsf{Sign}_2^*(S'_{i-1}, w_i)$
    $\Lambda_i := f_i(S_{i-1}, r_i)$
    $\Lambda'_i := h_i(S'_{i-1}, r'_i, w_i)$
    $i := i + 1$
    $w := w \cup m_i$
    Return $(\sigma_i, \Lambda_i, \Lambda'_i)$
```

Definition 2. [Existential Unforgeability with Leakage] *A signature scheme* Π^* *is* existentially unforgeable under adaptive chosen-message attacks in the presence of (continual) leakage *if* $\Pr[b = 1]$ *is negligible in the Experiment* Sign-Forge-Leak$_{\Pi^*}(\mathcal{A}, \kappa, \lambda)$ *for any efficient adversary* $\mathcal{A}$.

2.3 Bilinear Groups

Let $\mathsf{BGen}(\kappa)$ be a probabilistic bilinear group generator that outputs $(\mathbb{G}, \mathbb{G}_T, p, e, g)$ such that:

1. $\mathbb{G} = \langle g \rangle$ and $\mathbb{G}_T$ are (multiplicatively written) cyclic groups of prime order p with binary operations $\cdot$ and $\star$, respectively. The size of p is κ bits.
2. $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$ is a bilinear map that is:
   (a) bilinear: $\forall u, v \in \mathbb{G}$ and $\forall a, b \in \mathbb{Z}$, $e(u^a, v^b) = e(u, v)^{ab}$.
   (b) non-degenerate: $e(g, g) \neq 1$.

Such a group $\mathbb{G}$ is said to be a bilinear group the above properties hold. It is also required that the group operations in $\mathbb{G}$ and $\mathbb{G}_T$, and the map e are efficiently computable. The group $\mathbb{G}$ is called as *base group* and $\mathbb{G}_T$ as *target group*.

2.4 Generic Bilinear Group Model

The generic bilinear group (GBG) model [4] is an extension of the generic group model [23]. The encodings of the elements of $\mathbb{G}$ and $\mathbb{G}_T$ are given by random injective maps $\xi : \mathbb{Z}_p \to \Xi$ and $\xi_T : \mathbb{Z}_p \to \Xi_T$, respectively, where Ξ and Ξ_T are sets of bit-strings. The group operations in $\mathbb{G}$ and $\mathbb{G}_T$, and evaluation of the bilinear map e are performed by three public oracles $\mathcal{O}$, $\mathcal{O}_T$ and $\mathcal{O}_e$, respectively, defined as follows. For all $a, b \in \mathbb{Z}_p$

– $\mathcal{O}(\xi(a), \xi(b)) := \xi(a + b \bmod p)$
– $\mathcal{O}_T(\xi_T(a), \xi_T(b)) := \xi_T(a + b \bmod p)$
– $\mathcal{O}_e(\xi(a), \xi(b)) := \xi_T(ab \bmod p)$

We assume that $\Xi \cap \Xi_T = \phi$, the (fixed) generator g of $\mathbb{G}$ satisfies $g = \xi(1)$, and also the (fixed) generator g_T of $\mathbb{G}_T$ satisfies $g_T = e(g, g) = \xi_T(1)$.

2.5 Min-Entropy

Let X be a finite random variable with a probability distribution Pr. The *min-entropy* of X, denoted $\mathbf{H}_\infty(X)$, is defined as $\mathbf{H}_\infty(X) := -\log_2 \left(\max_x \Pr[X = x]\right)$. Min-entropy is a standard measure of the worst-case predictability of a random variable. Let Z be a random variable. The *average conditional min-entropy* of X given Z, denoted $\tilde{\mathbf{H}}_\infty(X \mid Z)$, is defined as

$$\tilde{\mathbf{H}}_\infty(X \mid Z) := -\log_2 \left(\mathop{\mathbb{E}}_{z \leftarrow Z} \left[\max_x \Pr[X = x \mid Z = z] \right] \right).$$

Average conditional min-entropy is a measure of the worst-case predictability of a random variable given a correlated random variable. The following result is due to [9].

Lemma 1. *Let $f : X \to \{0,1\}^{\lambda'}$ be a function on X. Then $\tilde{\mathbf{H}}_\infty(X \mid f(X)) \geq \mathbf{H}_\infty(X) - \lambda'$.*

The following result is a simple variant of the Schwartz-Zippel Lemma [22,24].

Lemma 2. [Schwartz-Zippel; min-entropy version] *Let $F \in \mathbb{Z}_p[X_1, \dots, X_n]$ be a non-zero polynomial of (total) degree at most d. Let P_i $(i = 1, \dots, n)$ be probability distributions on $\mathbb{Z}_p$ such that $\mathbf{H}_\infty(P_i) \geq \log p - \lambda'$, where $0 \leq \lambda' \leq \log p$. If $x_i \overset{P_i}{\leftarrow} \mathbb{Z}_p$ $(i = 1, \dots, n)$ are chosen independently, then $Pr[F(x_1, \dots, x_n) = 0] \leq 2^{\lambda'} \dfrac{d}{p}$.*

Proof. We prove the result by induction. When $n = 1$, the univariate polynomial F has at most d roots. Since $\mathbf{H}_\infty(P_1) \geq \log p - \lambda'$, we have $\Pr[F(x_1) = 0] \leq d\, 2^{-(\log p - \lambda')} = \frac{d}{p}\, 2^{\lambda'}$.

Let us now prove the result for the n-variables case assuming the result for the $(n-1)$-variables case. On writing F as a polynomial in X_1 with coefficients in $\mathbb{Z}_p[X_2, \dots, X_n]$, let i $(i \geq 1)$ be the degree of X_1 in the leading term and $F' \in \mathbb{Z}_p[X_2, \dots, X_n]$ be the leading coefficient. The probability

$$\begin{aligned} \Pr[F(x_1, \dots, x_n) = 0] \leq{} & \Pr[F(x_1, \dots, x_n) = 0 \mid F'(x_2, \dots, x_n) \neq 0] \\ & + \Pr[F'(x_2, \dots, x_n) = 0]. \end{aligned}$$

F' is now a non-zero polynomial, of degree at most $d - i$, in only $n - 1$ variables. By induction hypothesis we have $\Pr[F'(x_2, \dots, x_n) = 0] \leq \frac{d-i}{p}\, 2^{\lambda'}$. When $F'(x_2, \dots, x_n) \neq 0$, we have $\Pr[F(x_1, \dots, x_n) = 0] \leq \frac{i}{p}\, 2^{\lambda'}$ because degree of F in X_1 is i $(i \geq 1)$ and the distributions P_i $(i = 1, \dots, n)$ are independent. Hence $\Pr[F(x_1, \dots, x_n) = 0] \leq \frac{d}{p}\, 2^{\lambda'}$. Note that the parameter n does not appear in the above bound. □

Corollary 1. *If $\lambda' = (1 - \epsilon) \log p$ (for constant $\epsilon > 0$) in Lemma 2, then $Pr[F(x_1, \dots, x_n) = 0]$ is negligible (in $\log p$).*

3 Basic Signature Scheme

We now describe a signature scheme that is obtained from the Boneh-Boyen identity based encryption scheme (BB-IBE) [2]. This scheme is not yet known to be existentially unforgeable under adaptive chosen-message attacks (EUF-CMA) in the standard model. However, we are able to prove that the BB-signature scheme is EUF-CMA secure in the GBG model.

Let $\Pi_{\mathsf{BB}} = (\mathsf{KeyGen}_{\mathsf{BB}}, \mathsf{Sign}_{\mathsf{BB}}, \mathsf{Verify}_{\mathsf{BB}})$ be a signature scheme on the message space $\mathbb{Z}_p$ defined as follows:

1. $\mathsf{KeyGen}_{\mathsf{BB}}(\kappa)$: Compute $\mathbb{PP} := (\mathbb{G}, \mathbb{G}_T, p, e, g) \leftarrow \mathsf{BGen}(\kappa)$. Choose random $x, x_0, x_1 \leftarrow \mathbb{Z}_p$. Set $X := g^x$, $X_0 := g^{x_0}$, $X_1 := g^{x_1}$ and $X_T := e(g, X) = e(g, g)^x$. The public key is $pk := (\mathbb{PP}, X_0, X_1, X_T)$ and the secret key is $sk := X$.
2. $\mathsf{Sign}_{\mathsf{BB}}(sk, m)$: Choose a random $t \leftarrow \mathbb{Z}_p$. Set $\sigma := (sk \cdot (X_0 \cdot X_1^m)^t, g^t)$. Output the signature σ.
3. $\mathsf{Verify}_{\mathsf{BB}}(pk, m, \sigma)$: Let $\sigma = (\sigma_1, \sigma_2) \in \mathbb{G}^2$. Output the bit $b = 1$ (*valid*) if $X_T \star e(\sigma_2, X_0 \cdot X_1^m) = e(\sigma_1, g)$. Otherwise output $b = 0$ (*invalid*).

Theorem 1. *The signature scheme* Π_{BB} *is* EUF-CMA *secure in the generic bilinear group model.*

Proof. Let $\mathcal{A}$ be a q-query adversary that can break the security of Π_{BB}. By a q-query adversary we mean that $\mathcal{A}$ can make totally at most q group oracle and signing oracle queries. Let $q_{\mathcal{O}}$ be the total number of calls to the group oracles $\mathcal{O}$, $\mathcal{O}_T$ and $\mathcal{O}_e$, and q_{Ω} correspond to the number of calls to the signing oracle. We have $q_{\mathcal{O}} + q_{\Omega} \leq q$. As is typical for proofs in the generic group model, we bound the advantage of $\mathcal{A}$ against Π_{BB} by the success probability of $\mathcal{A}$ in the following game $\mathcal{G}$ (see [23,19,3]). $\mathcal{A}$ plays the game $\mathcal{G}$ with an algorithm $\mathcal{B}$.

Game $\mathcal{G}$: Let X, X_0, X_1, $\{T_i \,:\, 1 \leq i \leq q_{\Omega}\}$, $\{U_i \,:\, 1 \leq i \leq q_g,\ 0 \leq q_g \leq 2(q_{\mathcal{O}} + 1)\}$ and $\{V_i \,:\, 1 \leq i \leq q_{g_T},\ 0 \leq q_{g_T} \leq 2q_{\mathcal{O}}\}$ be indeterminates, and $\{m_i \,:\, 1 \leq i \leq q_{\Omega}\}$ be elements of $\mathbb{Z}_p$ chosen by $\mathcal{A}$. Intuitively, these indeterminates correspond to randomly chosen group elements in Π_{BB}, or more precisely their discrete logarithms. The indeterminates X, X_0, X_1 correspond to the quantities x, x_0, x_1, respectively. Note that $\mathcal{A}$ might query the group oracles with representations (bit-strings) not previously obtained from the group oracles. In order to accommodate this case we introduce the indeterminates U_i, V_i. The U_i $(1 \leq i \leq q_g)$ correspond to the elements of $\mathbb{G}$, whereas V_i $(1 \leq i \leq q_{g_T})$ correspond to the elements of $\mathbb{G}_T$. We denote the lists $\{T_i \,:\, 1 \leq i \leq q_{\Omega}\}$, $\{U_i \,:\, 1 \leq i \leq q_g\}$ and $\{V_i \,:\, 1 \leq i \leq q_{g_T}\}$ by $\{T\}$, $\{U\}$ and $\{V\}$, respectively.

$\mathcal{B}$ maintains two lists of pairs

$$\mathcal{L} = \{(F_{1,i}\,,\, \xi_{1,i}) \,:\, 1 \leq i \leq \tau_1\}, \tag{3}$$

$$\mathcal{L}_T = \{(F_{T,i}\,,\, \xi_{T,i}) \,:\, 1 \leq i \leq \tau_T\}, \tag{4}$$

such that, at step τ $(0 \leq \tau \leq q_{\mathcal{O}})$ in the game,

$$\tau_1 + \tau_T = \tau + 2q_{\Omega} + q_g + q_{g_T} + 4. \tag{5}$$

The entries $F_{1,i} \in \mathbb{Z}_p[X, X_0, X_1, \{U\}, \{T\}]$, $F_{T,i} \in \mathbb{Z}_p[X, X_0, X_1, \{U\}, \{V\}, \{T\}]$ are multivariate polynomials over $\mathbb{Z}_p$, whereas $\xi_{1,i}$, $\xi_{T,i}$ are bit-strings in the encoding sets Ξ (of $\mathbb{G}$) and Ξ_T (of $\mathbb{G}_T$), respectively. Intuitively, the polynomials in lists $\mathcal{L}$ and $\mathcal{L}_T$ correspond to elements of $\mathbb{G}$ and $\mathbb{G}_T$, respectively, that $\mathcal{A}$ will ever be able to compute or guess. In order to simplify the description, we view $\mathbb{Z}_p[X, X_0, X_1, \{U\}, \{T\}]$ as a subring of $\mathbb{Z}_p[X, X_0, X_1, \{U\}, \{V\}, \{T\}]$.

Initially, $\tau = 0, \tau_1 = 2q_{\Omega} + q_g + 3, \tau_T = q_{g_T} + 1$,

$$\begin{aligned}\mathcal{L} = \{ & (1, \xi_{1,1}), (X_0, \xi_{1,2}), (X_1, \xi_{1,3}), \{(U_i, \xi_{1,i+3}) : 1 \leq i \leq q_g\}, \\ & \{(X + (X_0 + m_i X_1)T_i, \xi_{1,2i+q_g+2}), (T_i, \xi_{1,2i+q_g+3}) : 1 \leq i \leq q_{\Omega}\} \}, \\ \mathcal{L}_T = \{ & X, \{(V_i, \xi_{T,i+1}) : 1 \leq i \leq q_{g_T}\} \}.\end{aligned}$$

The bit-strings $\xi_{1,i}$, $\xi_{T,i}$ are set to random distinct strings from Ξ and Ξ_T, respectively. We assume that there is some ordering (say, lexicographic ordering) among the strings in the sets Ξ and Ξ_T, so that given a string $\xi_{1,i}$ or $\xi_{T,i}$, it is possible to determine its index in the lists, if it exits.

The initial state of the two lists correspond to the group elements that $\mathcal{A}$ gets as input as part of the public parameters and the signatures obtained by $\mathcal{A}$ on the messages m_i of its choice. As previously mentioned, the polynomials U_i, V_i correspond to the group elements that $\mathcal{A}$ will guess in the actual interaction. Since $\mathcal{A}$ can query the group oracles with at most two new (guessed) elements and since it may also output at most two new elements from $\mathbb{G}$ as its forgery, we have $q_g + q_{g_T} \leq 2q_{\mathcal{O}} + 2$. Hence (5) can be simplified as (assuming $q_{\Omega} \geq 6$, without loss of generality)

$$\tau_1 + \tau_T \leq q_{\mathcal{O}} + 2q_{\Omega} + 2q_{\mathcal{O}} + 2 + 4 \leq 3(q_{\mathcal{O}} + q_{\Omega}) \leq 3q. \tag{6}$$

The game begins by $\mathcal{B}$ providing $\mathcal{A}$ with the initial τ_1 strings $\xi_{1,1}, \ldots, \xi_{1,\tau_1}$ from $\mathcal{L}$, and τ_T strings $\xi_{T,1}, \ldots, \xi_{T,\tau_T}$ from $\mathcal{L}_T$.

Group Operation: The calls made by $\mathcal{A}$ to the group oracles $\mathcal{O}$ and $\mathcal{O}_T$ are modeled as follows. For group operations in $\mathbb{G}$, $\mathcal{A}$ provides $\mathcal{B}$ with two operands (bit-strings) $\xi_{1,i}, \xi_{1,j}$ $(1 \leq i, j \leq \tau_1)$ in $\mathcal{L}$ and also specifies whether to multiply or divide them. $\mathcal{B}$ answers the query by first incrementing the counters $\tau_1 := \tau_1 + 1$ and $\tau := \tau + 1$, and provides $\mathcal{A}$ with the polynomial $F_{1,\tau_1} := F_{1,i} \pm F_{1,j}$. If $F_{1,\tau_1} = F_{1,k}$ for some $k < \tau_1$, then $\mathcal{B}$ sets $\xi_{1,\tau_1} := \xi_{1,k}$. Otherwise, ξ_{1,τ_1} is set to a random string distinct from those already present in $\mathcal{L}$. Also the pair $(F_{1,\tau_1}, \xi_{1,\tau_1})$ is appended to $\mathcal{L}$. Note that the (total) degree of the polynomials $F_{1,i}$ in $\mathcal{L}$ is at most two. Similarly, group operations in $\mathbb{G}_T$ are answered, appropriately updating the list $\mathcal{L}_T$ and the counters τ_T and τ.

Pairing: For a pairing operation, $\mathcal{A}$ queries $\mathcal{B}$ with two operands $\xi_{1,i}, \xi_{1,j}$ $(1 \leq i, j \leq \tau_1)$ in $\mathcal{L}$. $\mathcal{B}$ first increments $\tau_T := \tau_T + 1$ and $\tau := \tau + 1$, and then computes the polynomial $F_{T,\tau_T} := F_{1,i} \cdot F_{1,j}$. Again, if $F_{T,\tau_1} = F_{T,k}$ for some $k < \tau_T$, then $\mathcal{B}$ sets $\xi_{T,\tau_T} := \xi_{T,k}$. Otherwise, ξ_{T,τ_T} is set to a random string distinct from those already present in $\mathcal{L}_T$. Also the pair $(F_{T,\tau_T}, \xi_{T,\tau_T})$ is appended to $\mathcal{L}_T$. The degree of the polynomials $F_{T,i}$ in $\mathcal{L}_T$ is at most four.

When $\mathcal{A}$ terminates it outputs $(m, (\xi_{1,\alpha_1}, \xi_{1,\alpha_2})) \in \mathbb{Z}_p \times \mathcal{L} \times \mathcal{L}$ $(1 \le \alpha_1, \alpha_2 \le \tau_1)$. This corresponds to the "forgery" output by $\mathcal{A}$ in the actual interaction. Let the polynomials corresponding to ξ_{1,α_1} and ξ_{1,α_2} in $\mathcal{L}$ be F_{1,α_1} and F_{1,α_2}, respectively. After $\mathcal{A}$ terminates, $\mathcal{B}$ computes the polynomial

$$F_{1,\sigma} := X + F_{1,\alpha_2}(X_0 + mX_1) - F_{1,\alpha_1}. \tag{7}$$

Note that the degree of $F_{1,\sigma}$ is at most three. Next, $\mathcal{B}$ chooses random values x, x_0, x_1, $\{u\}$, $\{v\}$, $\{t\} \leftarrow \mathbb{Z}_p$ for the indeterminates X, X_0, X_1, $\{U\}$, $\{V\}$, $\{T\}$, respectively. Then it evaluates the polynomials in lists $\mathcal{L}$ and $\mathcal{L}_T$. $\mathcal{A}$ is said to have won the game $\mathcal{G}$ if:

1. $F_{1,i}(x, x_0, x_1, \{u\}, \{t\}) = F_{1,j}(x, x_0, x_1, \{u\}, \{t\})$ in $\mathbb{Z}_p$, for some two polynomials $F_{1,i} \ne F_{1,j}$ in $\mathcal{L}$.
2. $F_{T,i}(x, x_0, x_1, \{u\}, \{v\}, \{t\}) = F_{T,j}(x, x_0, x_1, \{u\}, \{v\}, \{t\})$ in $\mathbb{Z}_p$, for some two polynomials $F_{T,i} \ne F_{T,j}$ in $\mathcal{L}_T$.
3. $F_{1,\sigma}(x, x_0, x_1, \{u\}, \{t\}) = 0$ in $\mathbb{Z}_p$, and $m \ne m_i$ $\forall i$, $i = 1, \ldots, q_\Omega$.

This completes the description of the game $\mathcal{G}$.

We claim that the success probability of $\mathcal{A}$ in the actual EUF-CMA game is bounded above by its success probability in the above game $\mathcal{G}$. This is because of the following reasons:

– The conditions 1 and 2 above ensure that $\mathcal{A}$ will get to see only distinct group elements in the actual interaction. In other words, $\mathcal{A}$ is unable to cause *collisions* among group elements. As long as these two conditions are not satisfied, then the view of $\mathcal{A}$ is identical in the game $\mathcal{G}$ and the actual interaction. Hence if $\mathcal{A}$ is unable to provoke collisions, then adaptive strategies are no more powerful than non-adaptive ones (for more details, we refer to [19, Lemma 2 on pp. 12], also [23]). This observation allows us to choose group elements and their representations independently of the strategy of $\mathcal{A}$. Hence $\mathcal{A}$ specified the messages m_i at the beginning of the game $\mathcal{G}$ and also obtained the corresponding signatures. For the same reason, it also decided at the beginning itself on the representations it would guess. Note that the assumption that $\mathcal{A}$ would a priori decide the representations it would guess is only to simplify the description of the proof and it is not an inherent limitation.
– The condition 3 above ensures that the pair $(\xi_{1,\alpha_1}, \xi_{1,\alpha_2})$ is a valid forgery on a distinct message m.

We now compute the success probability of $\mathcal{A}$ in the game $\mathcal{G}$. The τ_1 polynomials $F_{1,i}$ in $\mathcal{L}$ have degree at most two. Note that $F_{1,i} \ne F_{1,j} \Leftrightarrow F_{1,i} - F_{1,j} \ne 0$ as polynomials. From Lemma 2 (with $\lambda' = 0$), the probability that two distinct polynomials in $\mathcal{L}$ evaluate to the same value for randomly and independently chosen values for the indeterminates is at most $\frac{2}{p}$. Summing up over at most $\binom{\tau_1}{2}$ distinct pairs (i, j), the probability that the condition 1 above holds is at most $\binom{\tau_1}{2} \cdot \frac{2}{p}$. Similarly, we have the probability that the condition 2 above holds is at most $\binom{\tau_2}{2} \cdot \frac{4}{p}$. The degree of the polynomial $F_{1,\sigma}$ in condition 3 is at most three. In order to apply Lemma 2, we need to prove that F_σ is not identically equal to the zero polynomial. We prove this fact in Lemma 3 below. Let $\Pr^{forge}_{\mathcal{A},\Pi_{\mathsf{BB}}}$ denote the advantage of the adversary $\mathcal{A}$ in computing a forgery against Π_{BB}. Then, assuming Lemma 3, we obtain from (6)

$$\Pr^{forge}_{\mathcal{A},\Pi_{\mathsf{BB}}} \leq \binom{\tau_1}{2} \cdot \frac{2}{p} + \binom{\tau_2}{2} \cdot \frac{4}{p} + \frac{3}{p} \leq \frac{2}{p}(\tau_1 + \tau_2)^2 \leq \frac{18q^2}{p}. \tag{8}$$

Hence if $q = poly(\log p)$, then $\Pr^{forge}_{\mathcal{A},\Pi_{\mathsf{BB}}}$ is negligible.

Lemma 3. *The polynomial $F_{1,\sigma} \in \mathbb{Z}_p[X, X_0, X_1, \{U\}, \{T\}]$ is non-zero.*

Proof. Any polynomial in $\mathcal{L}$ is obtained by either adding or subtracting two polynomials previously existing in the list. Hence we can write F_{1,α_1} and F_{1,α_2} in terms of polynomials present in $\mathcal{L}$ when it was was initialized at step $\tau = 0$ in the game $\mathcal{G}$. Note that initially $\mathcal{L}$ also includes the representations guessed by $\mathcal{A}$, in addition to the inputs.

$$\begin{aligned} F_{1,\alpha_1} = {} & c_1 + c_2X_0 + c_3X_1 + \textstyle\sum_{i=1}^{q_g} c_{4,i}U_i + \sum_{i=1}^{q_\Omega} c_{5,i}T_i \\ & + \textstyle\sum_{i=1}^{q_\Omega} c_{6,i}(X + (X_0 + m_iX_1)T_i), \end{aligned} \tag{9}$$

$$\begin{aligned} F_{1,\alpha_2} = {} & d_1 + d_2X_0 + d_3X_1 + \textstyle\sum_{i=1}^{q_g} d_{4,i}U_i + \sum_{i=1}^{q_\Omega} d_{5,i}T_i \\ & + \textstyle\sum_{i=1}^{q_\Omega} d_{6,i}(X + (X_0 + m_iX_1)T_i), \end{aligned} \tag{10}$$

where $c_j, d_j (j = 1, 2, 3), c_{j,i}, d_{j,i} (j = 4, 5, 6; 1 \leq i \leq q_\Omega) \in \mathbb{Z}_p$ are chosen by $\mathcal{A}$. We have two possible cases:

Case 1: $c_{6,i} = d_{6,i} = 0 \quad \forall i, 1 \leq i \leq q_\Omega$.

In this case, both F_{1,α_1} and F_{1,α_2} do not contain the indeterminate X. Hence the expression $F_{1,\alpha_2}(X_0 + mX_1) - F_{1,\alpha_1}$ in (7) is free of X. Therefore, in the polynomial $X + F_{1,\alpha_2}(X_0 + mX_1) - F_{1,\alpha_1}$, the coefficient of the term X is non-zero. Hence $F_{1,\sigma}$ is non-zero.

Case 2: $c_{6,k} \neq 0$ or $d_{6,k} \neq 0$ for some k, where $1 \leq k \leq q_\Omega$.

On substituting expressions from (9) and (10) into (7), we get that the coefficient of monomials $X_0^2T_i$, X_0T_i, X_1T_i in $F_{1,\sigma}$ are $d_{6,i}$, $d_{5,i} - c_{6,i}$, $m\, d_{5,i} - m_ic_{6,i}$, respectively, for $1 \leq i \leq q_\Omega$.

If $d_{6,k} \neq 0$, then the coefficient of $X_0^2T_k$ is non-zero, and hence $F_{1,\sigma} \neq 0$. Else, $c_{6,k} \neq 0$. We again have two cases: If $d_{5,k} \neq c_{6,k}$, then the coefficient of X_0T_k is non-zero. Or else, if $d_{5,k} = c_{6,k}$, then the coefficient of X_1T_k is non-zero, since $m \neq m_i\ \forall i, i = 1, \ldots, q_\Omega$. Hence in all cases we have $F_{1,\sigma}$ to be a non-zero polynomial. □

4 A Leakage-Resilient Signature Scheme

As previously mentioned in Section 2.2, any cryptographic scheme that does not maintain a stateful secret state is insecure against continual leakage. So is the case with the signature scheme $\mathsf{\Pi}_{\mathsf{BB}}$. We now describe a leakage-resilient version $\mathsf{\Pi}^*_{\mathsf{BB}}$ of $\mathsf{\Pi}_{\mathsf{BB}}$. We follow the techniques of [15] to adapt $\mathsf{\Pi}_{\mathsf{BB}}$ to a leakage setting. The basic idea is to store the secret key $X = g^x$ in two different parts of the memory as $(S_0 := g^{l_0}, S'_0 := g^{x-l_0})$ for a randomly chosen $l_0 \leftarrow \mathbb{Z}_p$. Accordingly, the $\mathsf{KeyGen}_{\mathsf{BB}}$ step of $\mathsf{\Pi}_{\mathsf{BB}}$ is modified to obtain the set-up stage $\mathsf{KeyGen}^*_{\mathsf{BB}}$ of $\mathsf{\Pi}^*_{\mathsf{BB}}$. The signature generation is now carried out as a two step process $\mathsf{Sign}^*_{\mathsf{BB1}}$ and $\mathsf{Sign}^*_{\mathsf{BB2}}$. During the i^{th} signature query, the two parts of the secret key (S_{i-1}, S'_{i-1}) are refreshed to obtain $(S_i := S_{i-1} \cdot g^{l_i}, S'_i := S'_{i-1} \cdot g^{-l_i})$, where $l_i \leftarrow \mathbb{Z}_p$. This is done in order to protect against continual leakage.

Let $\mathsf{\Pi}^*_{\mathsf{BB}} = (\mathsf{KeyGen}^*_{\mathsf{BB}}, \mathsf{Sign}^*_{\mathsf{BB1}}, \mathsf{Sign}^*_{\mathsf{BB2}}, \mathsf{Verify}^*_{\mathsf{BB}})$ be a stateful signature scheme on the message space $\mathbb{Z}_p$ defined as follows:

1. $\mathsf{KeyGen}^*_{\mathsf{BB}}(\kappa)$: Compute $\mathbb{PP} := (\mathbb{G}, \mathbb{G}_T, p, e, g) \leftarrow \mathsf{BGen}(\kappa)$. Choose random $x, x_0, x_1, l_0 \leftarrow \mathbb{Z}_p$. Set $X := g^x$, $X_0 := g^{x_0}$, $X_1 := g^{x_1}$ and $X_T := e(g, X) = e(g, g)^x$. The public key is $pk := (\mathbb{PP}, X_0, X_1, X_T)$ and the secret key is $sk^* := (S_0 := g^{l_0},\ S'_0 := g^{x-l_0} = X \cdot g^{-l_0}) \in \mathbb{G}^2$.
2. $\mathsf{Sign}^*_{\mathsf{BB1}}(S_{i-1}, m_i)$: Choose random $t_i, l_i \leftarrow \mathbb{Z}_p$. Set $S_i := S_{i-1} \cdot g^{l_i}$, $\sigma'_{1,i} := S_i \cdot (X_0 \cdot X_1^{m_i})^{t_i}$, and $\sigma'_{2,i} := g^{t_i}$.
3. $\mathsf{Sign}^*_{\mathsf{BB2}}(S'_{i-1}, (\sigma'_{1,i}, \sigma'_{2,i}, l_i))$: Set $S'_i := S'_{i-1} \cdot g^{-l_i}$ and $\sigma_i := (S'_i \cdot \sigma'_{1,i}, \sigma'_{2,i})$. Output the signature σ_i.
4. $\mathsf{Verify}^*_{\mathsf{BB}}(pk, m, \sigma)$: Let $\sigma = (\sigma_1, \sigma_2) \in \mathbb{G}^2$. Output the bit $b = 1$ (*valid*) if $X_T \star e(\sigma_2, X_0 \cdot X_1^m) = e(\sigma_1, g)$. Otherwise output $b = 0$ (*invalid*).

In steps 2 and 3 above, the index i keeps a count of the number of invocations (rounds) of the signing algorithm. For every $i \geq 1$, let $Y_i := \sum_{j=0}^{i} l_j$. It is easy to check that $S_i \cdot S'_i = g^{Y_i} \cdot g^{x-Y_i} = X$. We sometimes even refer to X as the secret key.

Note that $\mathsf{Sign}^*_{\mathsf{BB1}}$ requires four exponentiations and $\mathsf{Sign}^*_{\mathsf{BB2}}$ requires one. The total number of exponentiations needed for every signature invocation can be reduced from five to four if $\mathsf{Sign}^*_{\mathsf{BB1}}$ also passes on g^{l_i} to $\mathsf{Sign}^*_{\mathsf{BB2}}$. Hence only one extra exponentiation is needed when compared with the $\mathsf{Sign}_{\mathsf{BB}}$ step of $\mathsf{\Pi}_{\mathsf{BB}}$, which requires three.

For the sake of clarity, we would like to compare the various notations used in the signature scheme $\mathsf{\Pi}^*_{\mathsf{BB}}$ above with those in (1) corresponding to a generic stateful signature scheme $\mathsf{\Pi}^*$. The quantities r_i and w_i in (1) correspond to (l_i, t_i) and $(\sigma'_{1,i}, \sigma'_{2,i}, l_i)$ of $\mathsf{\Pi}^*_{\mathsf{BB}}$, respectively. The quantities S_i, S'_i and m_i denote the same things in both the cases. However, since the algorithm $\mathsf{Sign}^*_{\mathsf{BB2}}$ of $\mathsf{\Pi}^*_{\mathsf{BB}}$ does not generate any randomness, there is no analogue in $\mathsf{\Pi}^*_{\mathsf{BB}}$ for r'_i of (1). Accordingly, the leakage functions specified by an adversary to the signing oracle $\Omega^{\text{leak}}_{(S_{i-1}, S'_{i-1})}(m_i, f_i, h_i)$ would be of the form $f_i(S_{i-1}, (l_i, t_i))$ and $h_i(S'_{i-1}, (\sigma'_{1,i}, \sigma'_{2,i}, l_i))$.

First we show that Π^*_{BB} is secure in the GBG model when an adversary is not allowed to obtain leakage. The following lemma is a trivial consequence of the fact that the input/output behaviour of Π^*_{BB} and Π_{BB} are identical (c.f. Theorem 1).

Lemma 4. *The signature scheme* Π^*_{BB} *is* EUF-CMA secure *in the generic bilinear group model.*

The following theorem establishes the fact that the signature scheme Π^*_{BB} is resilient to (continual) leakage attacks in the GBG model if $\lambda \ll \frac{\log p}{2}$, where λ is the leakage parameter.

Theorem 2. *The signature scheme* Π^*_{BB} *is secure with leakage w.r.t. Definition 2 in the generic bilinear group model. The advantage of a q-query adversary who gets at most λ bits of leakage per each invocation of* $\mathsf{Sign}^*_{\mathsf{BB1}}$ *or* $\mathsf{Sign}^*_{\mathsf{BB2}}$ *is* $O\left(\frac{q^2}{p}2^{2\lambda}\right)$.

Proof. Let $\mathcal{A}$ be a q-query adversary that can break the security of Π^*_{BB}. By a q-query adversary $\mathcal{A}$ we mean that $\mathcal{A}$ can make totally at most q group oracle and signing oracle queries. Let $q_{\mathcal{O}}$ be the total number of calls to the group oracles $\mathcal{O}$, $\mathcal{O}_T$ and $\mathcal{O}_e$, and q_{Ω} correspond to the number of calls to the signing oracle. We have $q_{\mathcal{O}} + q_{\Omega} \leq q$. In the count $q_{\mathcal{O}}$, even the group oracle queries by leakage functions f_i, h_i specified by $\mathcal{A}$ are also included.

We first informally sketch the main ideas of the proof and then formalize these ideas. Let us try to see why the proof of security of Π^*_{BB} in the absence of any leakage (i.e. proof of Theorem 1) would not carry over as it is in the presence of leakage. In the non-leakage setting, while determining the probability of collision among distinct polynomials in conditions 1-3 on page 58, we substituted for each indeterminate an independent value chosen from an uniform distribution over $\mathbb{Z}_p$. But, when $\mathcal{A}$ has access to leakage functions $f_i(S_{i-1}, (l_i, t_i))$ and $h_i(S'_{i-1}, (\sigma'_{1,i}, \sigma'_{2,i}, l_i))$, then from its point of view the parameters t_i $(1 \leq i \leq q_{\Omega})$ are no longer uniformly distributed (though they are still independent). With some partial information about t_i, $\mathcal{A}$ can now cause collisions among polynomials with increased probability. Since each t_i is chosen independently and it can be leaked by only f_i, hence at most λ bits of t_i can be leaked. Apart from the values t_i, the only other "useful" information that leakage functions can provide is about the secret key $X = g^x$. This is because the parameters l_i themselves alone do not help $\mathcal{A}$ to output forgery since the signatures generated are independent of these randomly chosen values. Instead, $\mathcal{A}$ can very much use the leakages of l_i to compute, and eventually leak, the secret key X. Note that the leakage functions do not provide any additional information on the values x, x_0 or x_1.

We first bound the probability of the event that the secret key X is computed by some leakage function f_i or h_i. As long as this event has not occurred, then no bits of the secret key is leaked and the "only" additional information $\mathcal{A}$ has is about the values t_i. Clearly, the probability of this event depends on the leakage parameter λ. For instance, if the amount of leakage per invocation is not bounded, then during the first signature query itself, the adversary can leak the initial two shares of the secret key $S_0 = g^{l_0}$ and $S'_0 = X \cdot g^{-l_0}$ to recompute X. Finally, we determine the advantage of $\mathcal{A}$ conditioned on the event of the secret key X not being computed by any of the leakage functions.

Formally, we define E to be the event of computing (or guessing) the secret key $X = g^x$ by any of the leakage functions f_i or h_i $(1 \leq i \leq q_\Omega)$. Let $\overline{E}$ denote the complement of the event E, *Forgery* denote the event of $\mathcal{A}$ forging a signature on a new message, and $\Pr^{forge}_{\mathcal{A},\Pi^*_{\mathsf{BB}}} = \Pr[\text{Forgery}]$ denote the advantage of $\mathcal{A}$ in computing a forgery against Π^*_{BB}. We have

$$\Pr^{forge}_{\mathcal{A},\Pi^*_{\mathsf{BB}}} = \Pr[\text{Forgery}|E]\Pr[E] \ + \ \Pr[\text{Forgery}|\overline{E}]\Pr[\overline{E}].$$

Since $\Pr[\text{Forgery}|E]$, $\Pr[\overline{E}] \leq 1$, we obtain

$$\Pr^{forge}_{\mathcal{A},\Pi^*_{\mathsf{BB}}} \leq \Pr[E] + \Pr[\text{Forgery}|\overline{E}]. \tag{11}$$

We first bound the probability of the event E.

Lemma 5. $\Pr[E] \leq O\left(\frac{q^2}{p}2^{2\lambda}\right)$.

Proof. Let the adversary $\mathcal{A}$ play the following game $\mathcal{G}'$. Since the game $\mathcal{G}'$ is similar in nature to the game $\mathcal{G}$ in the proof of Theorem 1, we only briefly describe $\mathcal{G}'$. We use the notations introduced in the game $\mathcal{G}$. Let $\{L\}$ denote the list of indeterminates $\{L_i : 1 \leq i \leq q_\Omega\}$ that correspond to the values l_i in Π^*_{BB}.

Game $\mathcal{G}'$: For every leakage function $f_i(S_{i-1}, (l_i, t_i))$ and $h_i(S'_{i-1}, (\sigma'_{1,i}, \sigma'_{2,i}, l_i))$, $\mathcal{A}$ builds lists $\mathcal{L}^{f_i}$ and $\mathcal{L}^{h_i}$, respectively. These lists contain polynomial-bit string pairs. The polynomials are from $\mathbb{Z}_p[X, X_0, X_1, \{U\}, \{T\}, \{L\}]$ and the bit-strings are from the encoding set Ξ of the group $\mathbb{G}$. Intuitively, the polynomials in lists $\mathcal{L}^{f_i}$ and $\mathcal{L}^{h_i}$ correspond to the elements of group $\mathbb{G}$ that can be computed by f_i and h_i, respectively. Every polynomial in $\mathcal{L}^{f_i}$ is of the form

$$c_{1,i}L_i + c_{2,i}\sum_{j=0}^{i-1} L_j + c_{3,i}D_i, \tag{12}$$

where $c_{1,i}$, $c_{2,i}$, $c_{3,i} \in \mathbb{Z}_p$ are chosen by $\mathcal{A}$ and $D_i \in \mathbb{Z}_p[X, X_0, X_1, \{U\}, \{T\}]$ is in the list $\mathcal{L}$ (c.f. (3)). Every polynomial in $\mathcal{L}^{h_i}$ is of the form

$$d_{1,i}L_i + d_{2,i}\left(X - \sum_{j=0}^{i-1} L_j\right) + d_{3,i}\left(\left(\sum_{j=0}^{i} L_j\right) + (X_0 + m_i X_1)T_i\right) + d_{4,i}W_i, \tag{13}$$

where $d_{1,i}$, $d_{2,i}$, $d_{3,i}$, $d_{4,i}$, $m_i \in \mathbb{Z}_p$ are also chosen by $\mathcal{A}$ and $W_i \in \mathbb{Z}_p[X, X_0, X_1, \{U\}, \{T\}]$ is in the list $\mathcal{L}$.

When $\mathcal{A}$ terminates it outputs a polynomial F from the list $\mathcal{L}^{f_i}$ or $\mathcal{L}^{h_i}$, for some i. Intuitively, the polynomial F output by $\mathcal{A}$ corresponds to its guess of the secret key X. $\mathcal{A}$ is said to have won the game $\mathcal{G}'$ if

1. There is a collision in any of the lists $\mathcal{L}^{f_i}$ and $\mathcal{L}^{h_i}$, for some i ($1 \le i \le q_\Omega$).
2. $F - X = 0$ in $\mathbb{Z}_p$.

Note that the polynomials are now evaluated with values chosen from independent distributions with min-entropy $\log p - 2\lambda$. The reason for this will be shortly explained. This completes the description of the game $\mathcal{G}'$.

Technically speaking, $\mathcal{A}$ must also maintain lists $\mathcal{L}_T^{f_i}$ and $\mathcal{L}_T^{h_i}$ ($1 \le i \le q_\Omega$) that correspond to elements of the group $\mathbb{G}_T$ that can be computed by f_i and h_i. To simplify the discussion, we only describe collisions in the lists $\mathcal{L}^{f_i}$ and $\mathcal{L}^{h_i}$. Similar arguments apply for the lists $\mathcal{L}_T^{f_i}$ and $\mathcal{L}_T^{h_i}$. Since we compute $\Pr[E]$ only up to a constant factor, the additional advantage $\mathcal{A}$ obtains from collisions in $\mathcal{L}_T^{f_i}$ and $\mathcal{L}_T^{h_i}$ is implicitly included. However, working on the lines of the proof of Theorem 1, it is relatively straightforward to completely formalize the present discussion.

For similar reasons as given in the proof of Theorem 1, we have $\Pr[E]$ is bounded above by the success probability of $\mathcal{A}$ in the above game $\mathcal{G}'$. We particularly like to note the following. As observed in [1, pp. 691] and the references therein, even in the leakage setting adaptive strategies are no more powerful than non-adaptive ones.

Before computing the success probability of $\mathcal{A}$, we first show that $F - X$ is a non-zero polynomial. From Lemma 4 and Theorem 1, we know that Π^*_{BB} is secure without leakage. Hence the polynomial X (that corresponds to the secret key) cannot appear in the list $\mathcal{L}$, because this would otherwise imply that the secret key can be computed without access to leakage functions. A formal proof for this fact can be easily obtained on the lines of the proof of Lemma 3. Hence even when $c_{1,i} = c_{2,i} = 0$ in (12), the lists $\mathcal{L}^{f_i}$ cannot contain the polynomial X. If $c_{1,i} \neq 0$ or $c_{2,i} \neq 0$, then the polynomial in (12) will contain either L_i or L_{i-1}, or both. Hence the polynomial X cannot appear in any of the lists $\mathcal{L}^{f_i}$. In a similar way it can be seen that the lists $\mathcal{L}^{h_i}$ do not contain X. Hence $F - X$ is a non-zero polynomial of degree at most two.

Let us now determine the probability that the condition 1 above holds, i.e. the probability of collisions among distinct polynomials in any of the lists $\mathcal{L}^{f_i}$ and $\mathcal{L}^{h_i}$. In order to compute the probability, we evaluate the polynomials in (12) and (13) by choosing values from $\mathbb{Z}_p$ according to (independent) distributions with min-entropy at least $\log p - 2\lambda$. This is because $\mathcal{A}$ can obtain at most 2λ bits of leakage about l_i ($i = 0, \ldots, q_\Omega$), and at most λ bits of t_i ($i = 1, \ldots, q_\Omega$). From Lemma 1, the values l_i, t_i have min-entropy at least $\log p - 2\lambda$ in the view of $\mathcal{A}$. The total length of the lists $\mathcal{L}^{f_i}$, $\mathcal{L}^{h_i}$ is at most $O(q_\Omega + q_\mathcal{O}) = O(q)$. Hence there can be at most $O(q^2)$ pairs of distinct polynomials (of degree at most two) evaluating to the same value. From Lemma 2 (with $\lambda' = 2\lambda$), we obtain $\Pr[E] \le O\left(\frac{q^2}{p} 2^{2\lambda}\right)$. Since $F - X$ is a non-zero polynomial of degree at most two, the probability that $F - X$ evaluates to zero is at most $\frac{2}{p} 2^{2\lambda}$. This probability is also implicitly included in the above bound. □

We now determine the probability $\Pr[\text{Forgery} \mid \overline{E}]$ in (11).

Lemma 6. $\Pr[\text{Forgery} \,|\, \overline{E}] \leq \dfrac{18q^2}{p} 2^{\lambda}$.

Proof. Given that the event E has not occurred, the only meaningful leakage $\mathcal{A}$ can now obtain is that of t_i $(i = 1, \ldots, q_{\Omega})$. Since at most λ bits of t_i can leak (only by f_i), from the view point of $\mathcal{A}$ the values t_i have min-entropy at least $\log p - \lambda$. From Lemma 2 (with $\lambda' = \lambda$), the probability of collision among distinct polynomials in conditions 1-3 on page 58 is now increased by a factor of 2^{λ}. Hence, from (8), we obtain $\Pr[\text{Forgery} \,|\, \overline{E}] \leq \frac{18q^2}{p} 2^{\lambda}$. □

From (11) and Lemmas 5 and 6, we have $\Pr^{forge}_{\mathcal{A},\Pi^*_{\mathsf{BB}}} \leq O\left(\dfrac{q^2}{p} 2^{2\lambda}\right)$. This completes the proof of Theorem 2. □

Acknowledgements. We like to thank Jean-Sébastien Coron for his valuable comments on an early draft of this paper.

References

1. Aggarwal, D., Maurer, U.: The Leakage-Resilience Limit of a Computational Problem Is Equal to Its Unpredictability Entropy. In: Lee, D.H., Wang, X. (eds.) ASIACRYPT 2011. LNCS, vol. 7073, pp. 686–701. Springer, Heidelberg (2011)
2. Boneh, D., Boyen, X.: Efficient Selective-ID Secure Identity-Based Encryption Without Random Oracles. In: Cachin, C., Camenisch, J.L. (eds.) EUROCRYPT 2004. LNCS, vol. 3027, pp. 223–238. Springer, Heidelberg (2004)
3. Boneh, D., Boyen, X.: Short signatures without random oracles and the sdh assumption in bilinear groups. J. Cryptology 21(2), 149–177 (2008)
4. Boneh, D., Boyen, X., Goh, E.-J.: Hierarchical Identity Based Encryption with Constant Size Ciphertext. In: Cramer, R. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 440–456. Springer, Heidelberg (2005)
5. Boyle, E., Segev, G., Wichs, D.: Fully Leakage-Resilient Signatures. In: Paterson, K.G. (ed.) EUROCRYPT 2011. LNCS, vol. 6632, pp. 89–108. Springer, Heidelberg (2011)
6. Brakerski, Z., Kalai, Y.T., Katz, J., Vaikuntanathan, V.: Overcoming the hole in the bucket: Public-key cryptography resilient to continual memory leakage. In: FOCS, pp. 501–510. IEEE Computer Society (2010)
7. Coron, J.-S.: Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems. In: Koç, Ç.K., Paar, C. (eds.) CHES 1999. LNCS, vol. 1717, pp. 292–302. Springer, Heidelberg (1999)
8. Dodis, Y., Haralambiev, K., López-Alt, A., Wichs, D.: Efficient Public-Key Cryptography in the Presence of Key Leakage. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 613–631. Springer, Heidelberg (2010)
9. Dodis, Y., Ostrovsky, R., Reyzin, L., Smith, A.: Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. SIAM J. Comput. 38(1), 97–139 (2008)
10. Dziembowski, S., Faust, S.: Leakage-Resilient Cryptography from the Inner-Product Extractor. In: Lee, D.H., Wang, X. (eds.) ASIACRYPT 2011. LNCS, vol. 7073, pp. 702–721. Springer, Heidelberg (2011)

11. Dziembowski, S., Pietrzak, K.: Leakage-resilient cryptography. In: FOCS, pp. 293–302. IEEE Computer Society (2008)
12. Faust, S., Kiltz, E., Pietrzak, K., Rothblum, G.N.: Leakage-Resilient Signatures. In: Micciancio, D. (ed.) TCC 2010. LNCS, vol. 5978, pp. 343–360. Springer, Heidelberg (2010)
13. Ishai, Y., Sahai, A., Wagner, D.: Private Circuits: Securing Hardware against Probing Attacks. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 463–481. Springer, Heidelberg (2003)
14. Katz, J., Vaikuntanathan, V.: Signature Schemes with Bounded Leakage Resilience. In: Matsui, M. (ed.) ASIACRYPT 2009. LNCS, vol. 5912, pp. 703–720. Springer, Heidelberg (2009)
15. Kiltz, E., Pietrzak, K.: Leakage Resilient ElGamal Encryption. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 595–612. Springer, Heidelberg (2010)
16. Kocher, P.C.: 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)
17. Kocher, P.C., Jaffe, J., Jun, B.: Differential Power Analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999)
18. Malkin, T., Teranishi, I., Vahlis, Y., Yung, M.: Signatures Resilient to Continual Leakage on Memory and Computation. In: Ishai, Y. (ed.) TCC 2011. LNCS, vol. 6597, pp. 89–106. Springer, Heidelberg (2011)
19. Maurer, U.M.: Abstract Models of Computation in Cryptography. In: Smart, N.P. (ed.) Cryptography and Coding 2005. LNCS, vol. 3796, pp. 1–12. Springer, Heidelberg (2005)
20. Micali, S., Reyzin, L.: Physically Observable Cryptography. In: Naor, M. (ed.) TCC 2004. LNCS, vol. 2951, pp. 278–296. Springer, Heidelberg (2004)
21. Pietrzak, K.: A Leakage-Resilient Mode of Operation. In: Joux, A. (ed.) EUROCRYPT 2009. LNCS, vol. 5479, pp. 462–482. Springer, Heidelberg (2009)
22. Schwartz, J.T.: Fast probabilistic algorithms for verification of polynomial identities. J. ACM 27(4), 701–717 (1980)
23. Shoup, V.: Lower Bounds for Discrete Logarithms and Related Problems. In: Fumy, W. (ed.) EUROCRYPT 1997. LNCS, vol. 1233, pp. 256–266. Springer, Heidelberg (1997)
24. Zippel, R.: Probabilistic Algorithms for Sparse Polynomials. In: Ng, K.W. (ed.) EUROSAM 1979. LNCS, vol. 72, pp. 216–226. Springer, Heidelberg (1979)

Forward Secure Signatures on Smart Cards[⋆]

Andreas Hülsing, Christoph Busold, and Johannes Buchmann

Cryptography and Computeralgebra
Department of Computer Science
TU Darmstadt, Germany
{huelsing,buchmann}@cdc.informatik.tu-darmstadt.de,
christoph.busold@cased.de

Abstract. We introduce the forward secure signature scheme XMSS$^+$ and present an implementation for smart cards. It is based on the hash-based signature scheme XMSS. In contrast to the only previous implementation of a hash-based signature scheme on smart cards by Rohde et al., we solve the problem of on-card key generation. Compared to XMSS, we reduce the key generation time from $\mathcal{O}(n)$ to $\mathcal{O}(\sqrt{n})$, where n is the number of signatures that can be created with one key pair. To the best of our knowledge this is the first implementation of a forward secure signature scheme and the first full implementation of a hash-based signature scheme on smart cards. The resulting runtimes are comparable to those of RSA and ECDSA on the same device. This shows the practicality of forward secure signature schemes, even on constrained devices.

Keywords: forward secure signatures, smart cards, implementation, hash-based signatures.

1 Introduction

In 1997 Anderson introduced the idea of forward secure signature schemes (FSS) [3]. The idea behind FSS is the following: Even in the case of a key compromise, all signatures issued before the compromise should remain valid. This is an important property for all use cases where signatures have to stay valid for more than a short time period, including use cases like document signing or certificates. If for example a contract is signed, it is important that the signature stays valid for at least as long as the contract has some relevance. The solutions used today require the use of time stamps [13,14]. This introduces the requirement for a trusted third party and the overhead of requesting a time stamp for each signature. FSS in turn already provide this property and thereby abandon the need for time stamps. To fulfill the forward security property, a signature scheme has to be *key evolving*, meaning, the private key changes over time. The lifetime of a key pair is divided into time periods. While the public key stays the same, the secret key is updated at the end of each time period. So far, it was shown

⋆ Supported by grant no. BU 630/19-1 of the German Research Foundation (www.dfg.de).

that FSS can be efficiently implemented on PCs [6,11]. As for common signature schemes, to be usable in practice, FSS must be efficiently implementable on smart cards. This is even more important in the case of FSS, as it has to be ensured that the secret key is updated and the former secret key is deleted. So far there exists no implementation of FSS on smart cards.

A candidate FSS is the hash-based FSS XMSS [6] because of its strong security guarantees (see Section 2). Moreover, XMSS benefits from hardware acceleration for block ciphers, which is provided by many smart cards. A severe problem of most FSS, including XMSS, is the costly key generation. XMSS key generation requires time linear in the number of signatures that can be generated using the same key pair. While this might be tolerable on PCs, it makes key generation on smart cards impractical. The only existing implementation of a hash-based signature scheme on smart cards [22] does not include on-card key generation for this reason. But on-card key generation is necessary for most use cases that benefit from the forward security property. I.e. to guarantee non-repudiation in the case of document signing, a signature key pair has to be generated on the smart card and must never leave this secure environment.

Our contribution. In this paper we introduce XMSS$^+$ which is based on XMSS and present an implementation on an Infineon SLE78 smart card. While the strong security guarantees of XMSS are preserved, XMSS$^+$ key generation requires only time $\mathcal{O}(\sqrt{n})$, for a key pair, that can be used to sign n messages. Thereby we make on-card key generation practical. This means we present the first implementation of a forward-secure signature scheme on a smart card. At the same time, it is the first full (including key generation) smart card implementation of a hash-based signature scheme. To achieve this, we use the tree chaining technique [9] and improve the idea of distributed signature generation [7]. To improve the performance, we implemented all used (hash) function families based on AES and exploit the hardware acceleration provided by the card. Using our implementation, the generation of a key pair, that can be used to generate 2^{20} signatures, can be done in $22.2s$. For such a key pair, signature generation took less than $106ms$, verification took no more than $44ms$. These timings are of the same order of magnitude than the runtimes for RSA and ECDSA on the same card using the asymmetric crypto co-processor.

Organization. We start with a description of XMSS in Section 2. XMSS$^+$, that enables key generation, is presented and analyzed in Section 3. We describe our implementation and present parameters and runtimes in Section 4. Finally, we give a conclusion in Section 5.

2 The eXtended Merkle Signature Scheme XMSS

In this section we describe the FSS XMSS [6]. While there exist many proposals for FSS, including [1,2,4,10,15–17,19,23], XMSS is the only FSS where the forward security is based on minimal security assumptions. XMSS uses a function family $\mathcal{F}$ and a hash function family $\mathcal{H}$. It is provably forward secure in the standard model, if $\mathcal{F}$ is pseudorandom and $\mathcal{H}$ second preimage resistant. As current research suggests that these properties are not threatened by the existence of quantum computers, XMSS$^+$ is assumed to be resistant against quantum computer based attacks. We first give a high level overview. XMSS is build on a one-time signature scheme (OTS), a signature scheme where a key pair can only be used once. To obtain a many-time signature scheme, many OTS key pairs are used and their public keys are authenticated using a Merkle Tree. A Merkle Tree is a binary hash tree. The leaves of the tree are the hash values of the OTS public keys. The root of the Merkle Tree is the XMSS public key. To overcome the need of storing all OTS key pairs, they are generated using a pseudorandom generator (PRG). We start the detailed description with the parameters used by XMSS, afterwards we give a description of the building blocks, namely, the Winternitz-OTS, the XMSS Tree, the leaf construction, and the PRG. Then we describe the algorithms for key generation, signature generation and verification. In the following we write log for $\log_2$ and $x \xleftarrow{\$} X$ if the value x is chosen uniformly at random from the set X.

Parameters. For security parameter $n \in \mathbb{N}$, XMSS uses a pseudorandom function family $\mathcal{F}_n = \{\mathrm{F}_K : \{0,1\}^n \rightarrow \{0,1\}^n | K \in \{0,1\}^n\}$, and a second preimage resistant hash function H, chosen uniformly at random from the family $\mathcal{H}_n = \{\mathrm{H}_K : \{0,1\}^{2n} \rightarrow \{0,1\}^n | K \in \{0,1\}^n\}$. It is parameterized by the message length $m \in \mathbb{N}$, the tree height $h \in \mathbb{N}$, the BDS parameter $k \in \mathbb{N}, k < h, k-h$ is even, and the Winternitz parameter $w \in \mathbb{N}, w > 1$. XMSS can be used to sign 2^h message digests of m bits. The Winternitz parameter w allows for a trade off between signature generation time and signature size. The BDS parameter k allows for a time-memory trade-off for the signature generation. Those parameters are publicly known.

Winternitz OTS. XMSS uses the Winternitz-OTS (W-OTS) from [5]. W-OTS uses the function family $\mathcal{F}_n$ and a value $X \in \{0,1\}^n$ that is chosen during XMSS key generation. For $K, X \in \{0,1\}^n$, $e \in \mathbb{N}$, and $\mathrm{F}_K \in \mathcal{F}_n$ we define $\mathrm{F}_K^e(X)$ as follows. We set $\mathrm{F}_K^0(X) = K$ and for $e > 0$ we define $K' = \mathrm{F}_K^{e-1}(X)$ and $\mathrm{F}_K^e(X) = \mathrm{F}_{K'}(X)$. Also, define

$$\ell_1 = \left\lceil \frac{m}{\log(w)} \right\rceil, \quad \ell_2 = \left\lfloor \frac{\log(\ell_1(w-1))}{\log(w)} \right\rfloor + 1, \quad \ell = \ell_1 + \ell_2.$$

The secret signature key of W-OTS consists of ℓ n-bit strings sk_i, $1 \leq i \leq \ell$. The generation of the sk_i will be explained later. The public verification key is computed as

$$\mathsf{pk} = (\mathsf{pk}_1, \ldots, \mathsf{pk}_\ell) = (\mathrm{F}_{\mathsf{sk}_1}^{w-1}(X), \ldots, \mathrm{F}_{\mathsf{sk}_\ell}^{w-1}(X)),$$

with F^{w-1} as defined above. W-OTS signs messages of binary length m. They are processed in base w representation. They are of the form $M = (M_1 \ldots M_{\ell_1})$,

$M_i \in \{0, \ldots, w-1\}$. The checksum $C = \sum_{i=1}^{\ell_1}(w - 1 - M_i)$ in base w representation is appended to M. It is of length ℓ_2. The result is a sequence of ℓ base w numbers, denoted by $(T_1, \ldots, T_\ell)$. The signature of M is

$$\sigma = (\sigma_1, \ldots, \sigma_\ell) = (\mathrm{F}_{\mathsf{sk}_1}^{T_1}(X), \ldots, \mathrm{F}_{\mathsf{sk}_\ell}^{T_\ell}(X)).$$

It is verified by constructing $(T_1 \ldots, T_\ell)$ and checking

$$(\mathrm{F}_{\sigma_1}^{w-1-T_1}(X), \ldots, \mathrm{F}_{\sigma_\ell}^{w-1-T_\ell}(X)) \stackrel{?}{=} (\mathsf{pk}_1, \ldots, \mathsf{pk}_\ell).$$

The sizes of signature, public, and secret key are ℓn bits. For more detailed information see [5].

XMSS Tree. The XMSS tree utilizes the hash function H. The XMSS tree is a binary tree of height h. It has $h + 1$ levels. The leaves are on level 0. The root is on level h. The nodes on level j, $0 \le j \le h$, are denoted by $N_{i,j}$, $0 \le i < 2^{h-j}$. To construct the tree, h bit masks $B_j \in \{0,1\}^{2n}$, $0 < j \le h$, are used. $N_{i,j}$, for $0 < j \le h$, is computed as

$$N_{i,j} = \mathrm{H}((N_{2i,j-1} || N_{2i+1,j-1}) \oplus B_j).$$

Leaf Construction. The leaves of the XMSS tree are the hash values of the W-OTS public keys. To avoid the need of a collision resistant hash function, another XMSS tree is used to construct the leaves. It is called L-tree. The ℓ leaves of an L-tree are the ℓ bit strings $(\mathsf{pk}_0, \ldots, \mathsf{pk}_\ell)$ from the corresponding verification key. As ℓ is not necessarily a power of 2, there might not be sufficiently many leaves to get a complete binary tree. Therefore the construction is modified. A left node that has no right sibling is lifted to a higher level of the L-tree until it becomes the right sibling of another node. In this construction, the same hash function as above but new bitmasks are used. The bitmasks are the same for all L-trees. As L-trees have height $\lceil \log \ell \rceil$, additional $\lceil \log \ell \rceil$ bitmasks are required.

Pseudorandom Generator. The W-OTS key pairs are generated using two pseudorandom generators (PRG). The stateful forward secure PRG $\textsc{FsGen} : \{0,1\}^n \to \{0,1\}^n \times \{0,1\}^n$ is used to generate one seed value per W-OTS key-pair, using the function family $\mathcal{F}_n$. Then the seed is expanded to the ℓ W-OTS secret key bit strings using $\mathcal{F}_n$. $\textsc{FsGen}$ starts from a uniformly random state $S_0 \xleftarrow{\$} \{0,1\}^n$. On input of a state S_i, $\textsc{FsGen}$ generates a new state S_{i+1} and a pseudorandom output R_i:

$$\textsc{FsGen}(S_i) = (S_{i+1} || R_i) = (\mathrm{F}_{S_i}(0) || \mathrm{F}_{S_i}(1)).$$

The output R_i is used to generate the ith W-OTS secret key $(\mathsf{sk}_1, \ldots, \mathsf{sk}_\ell)$:

$$\mathsf{sk}_j = \mathrm{F}_{R_i}(j-1), 1 \le j \le \ell.$$

Key Generation. The key generation algorithm takes as input all of the above parameters. Then the whole XMSS Tree has to be constructed to obtain the value of the root node. We now detail this procedure. First, the bitmasks $(B_1, \dots, B_{h+\lceil \log \ell \rceil})$ and the value X are chosen uniformly at random. Then, the initial state of FsGen, S_0 is chosen uniformly at random and a copy of it is stored as part of the secret key SK. The tree is constructed using the TreeHash algorithm, listed as Algorithm 1 below. Starting with an empty stack Stack and S_0, all 2^h leaves are successively generated and used as input to the TreeHash algorithm to update Stack. This is done by evaluating FsGen on the current state S_i, obtaining R_i and replacing S_i with S_{i+1}. Then R_i is used to compute the W-OTS public key, which in turn is used to compute the corresponding leaf using an L-tree. The leaf and the current Stack are then used as input for the TreeHash algorithm to obtain an updated Stack. The W-OTS key pair and R_i are deleted. After all 2^h leaves were processed by TreeHash, the only value on Stack is the root of the tree, which is stored in the public key PK.

Algorithm 1. TreeHash

```
Input: Stack Stack, node $N_1$
Output: Updated stack Stack

1. While top node on Stack has same height as $N_1$ do
   (a) $t \leftarrow N_1.height() + 1$
   (b) $N_1 \leftarrow \mathrm{H}((\mathsf{Stack}.pop() || N_1) \oplus B_t)$
2. Stack.push($N_1$)
3. Return Stack
```

The XMSS signature generation algorithm uses as subroutine the BDS algorithm [8] that is explained there. The BDS algorithm uses a state $\mathsf{State}_{\mathrm{BDS}}$ which is initialized during the above computation of the root. For details see [8]. The initial XMSS secret key $\mathsf{SK} = (S_0, \mathsf{State}_{\mathrm{BDS}})$ contains the initial states of FsGen and the BDS algorithm. The XMSS public key consists of the bitmasks $(B_1, \dots, B_{h+\lceil \log \ell \rceil})$, the value X, and the root of the tree. As shown in [6], key generation requires $2^h(\ell + 1)$ evaluations of H and $2^h(2 + \ell(w + 1))$ evaluations of functions from $\mathcal{F}_n$.

Signature Generation. The signature generation algorithm takes as input a message M, the secret key SK and the index i. It outputs an updated secret key SK' and a signature Σ on the message M. To sign the ith message (we start counting from 0), the ith W-OTS key pair is used. The signature $\Sigma = (i, \sigma, \mathsf{Auth})$ contains the index i, the W-OTS signature σ, and the authentication path for the leaf $N_{0,i}$. The authentication path is the sequence $\mathsf{Auth} = (\mathsf{Auth}_0, \dots, \mathsf{Auth}_{h-1})$ of the siblings of all nodes on the path from $N_{0,i}$ to the root. Figure 1 shows the authentication path for leaf i. We now explain how a signature is generated. On input of the ith message, SK contains the ith state S_i of FsGen. So, FsGen is evaluated on S_i to obtain S_{i+1}, which becomes the updated secret key, and R_i. R_i is used to generate the ith W-OTS secret key, which in turn is used to generate the one-time signature σ on M. Then the authentication path is computed using the BDS tree traversal algorithm from [8] which we explain next.

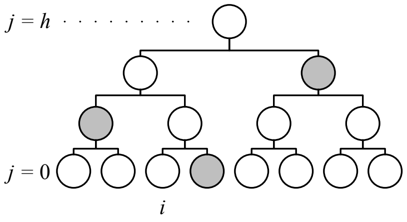


Fig. 1. The authentication path for leaf i

The BDS algorithm uses TreeHash to compute the nodes of the authentication path. The computation of a node on level i takes 2^i leaf computations and 2^i evaluations of TreeHash. If all this computation is done when the authentication path is needed, the computation of an authentication path requires $2^h - 1$ leaf computations and evaluations of TreeHash in the worst case. The BDS algorithm reduces the worst case signing time to $(h-k)/2$ leaf computations and evaluations of TreeHash. More specifically, the BDS algorithm does three things. First, it uses the fact that a node that is a left child can be computed from values that occurred in an authentication path before, spending only one evaluation of H. Second, it stores the right nodes from the top k levels of the tree during key generation. So these nodes, that are most expensive to compute, do not have to be computed again during signature generation. Third, it distributes the computations for right child nodes among previous signature generations. This is done, using one instance of TreeHash per tree level. The computation of the next right node on a level starts, when the last computed right node becomes part of the authentication path. The BDS algorithm uses a state $\mathsf{State}_{\mathrm{BDS}}$ of $2(h-k)$ states of FsGen and at most $\left(3h + \left\lfloor \frac{h}{2} \right\rfloor - 3k - 2 + 2^k\right)$ tree nodes. $\mathsf{State}_{\mathrm{BDS}}$ is initialized during key generation. After initialization, it contains the right nodes on the k top levels, the first authentication path (for $N_{0,0}$) and the second right node on each level. To compute the authentication paths, the BDS algorithm spends only $(h-k)/2$ leaf computations and evaluations of TreeHash to update its state per signature. This update is done such that at the end of the ith signature generation, $\mathsf{State}_{\mathrm{BDS}}$ already contains the authentication path for leaf $i+1$. For more details see [8].

Signature Verification. The signature verification algorithm takes as input a signature $\Sigma = (i, \sigma, \mathsf{Auth})$, the message M and the XMSS public key PK. To verify the signature, the values $(T_0, \ldots, T_\ell)$ are computed as described in the

W-OTS signature generation, using M. Then the ith verification key is computed using the formula

$$(\mathsf{pk}_1, \ldots, \mathsf{pk}_\ell) = (\mathrm{F}_{\sigma_1}^{w-1-T_1}(X), \ldots, \mathrm{F}_{\sigma_\ell}^{w-1-T_\ell}(X)).$$

The corresponding leaf $N_{0,i}$ of the XMSS tree is constructed using an L-tree. This leaf and the authentication path are used to compute the path $(P_0, \ldots, P_h)$ to the root of the XMSS tree, where $P_0 = N_{0,i}$ and

$$P_j = \begin{cases} \mathrm{H}((P_{j-1}||\mathsf{Auth}_{j-1}) \oplus B_j), & \text{if } \lfloor i/2^j \rfloor \equiv 0 \mod 2 \\ \mathrm{H}((\mathsf{Auth}_{j-1}||P_{j-1}) \oplus B_j), & \text{if } \lfloor i/2^j \rfloor \equiv 1 \mod 2 \end{cases}$$

for $0 \leq j \leq h$. If P_h is equal to the root of the XMSS tree given in the public key, the signature is accepted. Otherwise, it is rejected.

3 XMSS$^+$: On-Card Key Generation

In [22], a hash-based signature scheme similar to XMSS is implemented on smart cards. But they did not implement on-card key generation, because of the heavy computations required. In this section we introduce XMSS$^+$, which allows for fast on-card key generation. The techniques used are based on the tree chaining technique introduced in [9] and distributed signature generation from [7]. The basic idea is the following. To obtain an instance of XMSS$^+$ that can be used to make 2^h signatures, we use two levels of XMSS key pairs with height $h/2$ instead of one key pair with height h: One key pair on the upper level ($\mathcal{U}$) of height $h/2$ is used to sign the roots of $2^{h/2}$ key pairs on the lower level ($\mathcal{L}s$) of height $h/2$. The root of $\mathcal{U}$ becomes the public key and the $\mathcal{L}s$ are used to sign the messages. During key generation, $\mathcal{U}$ and the first $\mathcal{L}$ are generated. The generation of the remaining $\mathcal{L}s$ is distributed among signature generations. As a result, the time to generate a key pair, that can be used to sign 2^h messages, goes down from $\mathcal{O}(2^h)$ to $\mathcal{O}(2^{h/2})$.

A signature always contains the current index, the signature of the message using the current $\mathcal{L}$, and the signature of the root of $\mathcal{L}$ under $\mathcal{U}$. To decrease the worst case signing time, the authors of [7] propose to equally distribute the costs for signing the roots of the $\mathcal{L}s$ among the message signatures. For XMSS$^+$ we propose a new approach to distribute these costs. We use the observation that the BDS algorithm does not always use all updates it receives. These unused updates can be used to compute the signatures of the roots from the $\mathcal{L}s$. Thereby we reduce the worst case signing time, again. We use the same bit masks and the same X value for all trees. Thereby the public key size is reduced, as it contains less bit masks. To generate the secret keys, we select a random initial state for FsGen for each key pair, just in time. Now we describe the key generation, signature generation and signature verification algorithms in detail.

Key generation. The XMSS$^+$ key generation algorithm takes as inputs the security parameter n, the message length m, the hash function H, the function family $\mathcal{F}$, and the overall height h, h is even. We set the internal tree height $h' = h/2$. In contrast to the last section, it takes two Winternitz parameters w_u, w_l and two BDS parameters k_u, k_l such that $h' - k_i$ is even for $i \in \{l, u\}$ and $(h' - k_u)/2 + 1 \leq 2^{h'-k_l+1}$. As for XMSS, the bitmasks and the X are chosen uniformly at random, but this time $h' + \max\{\log \ell_u, \log \ell_l\}$ bitmasks are chosen. Both, the bitmasks and the X are used for both levels. Then the two XMSS key pairs $\mathcal{L}$ and $\mathcal{U}$ are generated. This is done as described in the last section. For $\mathcal{L}$, w_l, k_l, and the message length m are used. For $\mathcal{U}$, w_u and k_u are used. The message length for $\mathcal{U}$ is n, because this is the size of the root nodes of the $\mathcal{L}s$. Next, the root of $\mathcal{L}$ is signed using the first W-OTS keypair of $\mathcal{U}$. Then, a FsGen state for the next $\mathcal{L}$ is chosen uniformly at random, and a new TreeHash stack Stack_{next} is initialized.

The XMSS$^+$ secret key SK consists of the two FsGen states S_l and S_u and the BDS states $\mathsf{State}_{\mathrm{BDS},l}$ and $\mathsf{State}_{\mathrm{BDS},u}$ for $\mathcal{U}$ and $\mathcal{L}$ and the signature on the root of $\mathcal{L}$. Additionally, it contains a FsGen state S_n, a TreeHash stack Stack_{next} and a BDS state $\mathsf{State}_{\mathrm{BDS},n}$ for the next $\mathcal{L}$. The public key PK consists of the $h' + \max\{\log \ell_1, \log \ell_2\}$ bitmasks, the value X and the root of $\mathcal{U}$.

Signature generation. The signature generation algorithm takes as input a message M, the secret key SK, and the index i. First, M is signed. This is done as described in the last section, using S_l and $\mathsf{State}_{\mathrm{BDS},l}$ as secret key for $\mathcal{L}$ and $i \bmod 2^{h'}$ as index. During this signature generation, BDS receives $(h' - k_l)/2$ updates. If not all of these updates are used to update $\mathsf{State}_{\mathrm{BDS},l}$, the remaining updates are used to update $\mathsf{State}_{\mathrm{BDS},u}$. Then one leaf of the next lower tree is computed and used as input for TreeHash to update Stack_{next}. The signature $\Sigma = (\sigma_u, \mathsf{Auth}_u, \sigma_l, \mathsf{Auth}_l, i)$ contains the one-time signatures from $\mathcal{U}$ and $\mathcal{L}$ and the two authentication paths, as well as the index i.

If $i \bmod 2^{h'} = 2^{h'} - 1$ the last W-OTS key pair of the current $\mathcal{L}$ was used. In this case, Stack_{next} now contains the root of the next $\mathcal{L}$. Now, $\mathcal{U}$ is used to sign this root. The key pair consists of S_u and $\mathsf{State}_{\mathrm{BDS},u}$. The used index is $\lceil i/2^{h'} \rceil$. In contrast to the signing algorithm from the last section, BDS receives no updates at this time. The updates needed to compute the next authentication path are received during the next $2^{h'}$ message signatures. In SK $\mathsf{State}_{\mathrm{BDS},l}$, S_l, and the signature of the root of the $\mathcal{L}$ are replaced by $\mathsf{State}_{\mathrm{BDS},n}$, S_n and the new computed signature, respectively. Afterwards, the data structures for the next $\mathcal{L}$ are initialized and used to replace the ones in SK.

Signature verification. The signature verification algorithm takes as input a signature $\Sigma = (\sigma_u, \mathsf{Auth}_u, \sigma_l, \mathsf{Auth}_l, i)$, the message M and the public key PK. To verify the signature, M and σ_l are used to construct the corresponding W-OTS public key, and then the corresponding leaf node. This leaf node, Auth_l and the index $j = i \bmod 2^{h'}$ are used to compute the root of $\mathcal{L}$. This root in turn, is used together with σ_u to compute the W-OTS public key and the corresponding leaf node of $\mathcal{U}$. This leaf node, Auth_u and the index $j = \lfloor i/2^{h'} \rfloor$ are used to compute a root for $\mathcal{U}$. The root computations are done as described in the last section. If the resulting root equals the root node included in the public key, the signature is accepted and rejected otherwise.

3.1 Analysis

In the following we provide an analysis of XMSS$^+$. We show that the distributed authentication path computation works and revisit the security of the scheme. We start with key and signature sizes and the runtimes of the algorithms. A theoretical comparison with XMSS will be included in the full version of this paper.

Sizes and Runtimes. First we look at the sizes. The signature size grows by the size of one W-OTS signature and is $(h+\ell_u+\ell_l)n$ bits. The public key size slightly decreases, as the number of bitmasks decreases and is $(h+2\max\{\log \ell_u, \log \ell_l\}+2)n$ bits. The secret key stays about the same size, depending on the parameter choices, and is at most $(7.5h-7k_l-5k_u+2^{k_l}+2^{k_u}+\ell_u)n$ bits. For the runtimes we only look at the worst case times and get the following. The key generation time is reduced to $2^{h/2}(\ell_u+\ell_l+2)t_{\mathrm{H}}+2^{h/2}(4+\ell_u(w_u+1)+\ell_l(w_l+1))t_{\mathrm{F}}$, where t_{H} and t_{F} denote the runtimes of one evaluation of H and F, respectively. The worst case signing time also decreases because the trees are smaller and requires less than $\max_{i\in\{l,u\}}\{(((h'-k_l+2)/2)\cdot(h'-k_i+\ell_i)+h')t_{\mathrm{H}}+(((h'-k_l+4)/2)\cdot(\ell_i(w_i+1))+h'-k_l)t_{\mathrm{F}}\}$ (Recall that $h'=h/2$). Signature verification increases by the costs of verifying one W-OTS signature and computing the corresponding leaf. It requires $(\ell_u+\ell_l+h)t_{\mathrm{H}}+(\ell_u w_u+\ell_l w_l)t_{\mathrm{F}}$.

Correctness. In the following we show, that the unused updates from $\mathcal{L}$ suffice to compute the authentication paths and to sign the next root. For the computation of the ith authentication path Auth_i in $\mathcal{U}$ and the signature on the $(i+1)$th root, all unused updates from the $(i-1)$th $\mathcal{L}$ can be used. The signature algorithm spends $(h'-k_l)/2$ updates per signature. Hence, the BDS algorithm receives $(h'-k_l)2^{h'-1}$ updates while the $(i-1)$th $\mathcal{L}$ is used. For all authentication paths of $\mathcal{L}$, the BDS algorithm has to compute all right nodes of the tree, that are on a height $< h'-k_l$, besides the two first right nodes on every height as these nodes are already stored during initialization. The number of required updates for $2\leq k_l\leq h'$ is

$$\sum_{i=0}^{h'-k_l-1}(2^{h'-i-1}-2)2^i=(h'-k_l)2^{h'-1}-2^{h'-k_l+1}$$

so there are $(h'-k_l)2^{h'-1}-(h'-k_l)2^{h'-1}+2^{h'-k_l+1}=2^{h'-k_l+1}$ unused updates. As $(h'-k_u)/2+1\leq 2^{h'-k_l+1}$, the BDS algorithm for the $\mathcal{U}$ receives all $(h'-k_u)/2$ updates to compute Auth_i before it is needed and one update is left for the signature on the next root. Doing the same computation for $k_l=0$ there are even more $(3\cdot 2^{h'-1})$ unused updates. For $k_l=h'$, it follows from $(h'-k_u)/2+1\leq 2^{h'-k_l+1}$ that $k_u=h'$ and therefore all nodes of both trees are stored.

Security. In [6], an exact proof is given which shows that XMSS is forward secure, if $\mathcal{F}$ is a pseudorandom function family and $\mathcal{H}$ a second preimage resistant hash function family. The tree chaining technique corresponds to the product composition from [19]. In [19] the authors give an exact proof for the forward security of the product composition if the underlying signature schemes are forward secure. It is straight forward to combine both security proofs to obtain an exact proof for the forward security of XMSS$^+$.

4 Implementation

In this section we present our smart card implementation. First we give a description of our implementation. Then we present our results and give a comparison with XMSS, RSA and ECDSA. At the end of the section we discuss an issue regarding the non-volatile memory (NVM).

Implementation Details. For the implementation we use an Infineon SLE78 CFLX4000PM offering 8 KB RAM and 404 KB NVM. Its core consists of a 16-bit CPU running at 33 MHz. Besides other peripherals, it provides a True Random Number Generator (TRNG), a symmetric and an asymmetric crypto co-processor. We use the hardware accelerated AES implementation of the card to implement the function families $\mathcal{F}$ and $\mathcal{H}$. As proposed in [6], we use plain AES for $\mathcal{F}$. To implement $\mathcal{H}$ we build a compression function using the Matyas-Meyer-Oseas construction [20] and iterate it using the Merkle-Darmgard construction [12,21]. As the input size of $\mathcal{H}$ is fixed, we do not require M-D strengthening. Figure 2 shows the whole construction. As shown there, the construction requires two AES evaluations per evaluation of $\mathrm{H}_K \in \mathcal{H}$. All random inputs of the scheme are generated using the TRNG. Besides XMSS$^+$, we also implemented XMSS for comparison.

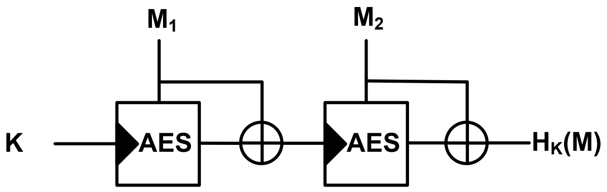


Fig. 2. Construction of $\mathcal{H}$ using AES with the Matyas-Meyer-Oseas construction in M-D Mode

Results. Tables 1 and 2 show the runtimes of our implementation with different parameter sets. We use the same k and w for both trees. The last column shows the security level for the given parameter sets. Following the updated heuristic of Lenstra and Verheul [18] the configurations with a security level of 81 (85, 86) bits are secure until the year 2019 (2025, 2026). In Appendix A we explain how the security level is computed. Please note that these numbers represent a lower bound on the provable security level. A successful attack would still require an adversary to either find a second preimage in a 128 bit hash function or to launch a successful key retrieval attack on AES 128. This would result in 128 bit security for all parameter sets. In Table 1, the signature time is the worst case time over all signatures of one key pair. The secret key size in the table differs from the values we would obtain using the theoretical formulas from the last section. This is because it includes all data that has to be stored on the card to generate signatures, including the bitmasks and X.

Table 1. Results for XMSS and XMSS$^+$ for message length $m = 256$ on an Infineon SLE78. We use the same k and w for both trees. b denotes the security level in bits. The signature times are worst case times.

				Timings (ms)			Sizes (byte)			
Scheme	h	k	w	KeyGen	Sign	Verify	Secret key	Public key	Signature	b
XMSS$^+$	16	2	4	5,600	106	25	3,760	544	3,476	85
XMSS$^+$	16	2	8	5,800	105	21	3,376	512	2,436	81
XMSS$^+$	16	2	16	6,700	118	22	3,200	512	1,892	71
XMSS$^+$	16	2	32	10,500	173	28	3,056	480	1,588	54
XMSS$^+$	20	4	4	22,200	106	25	4,303	608	3,540	81
XMSS$^+$	20	4	8	22,800	105	21	3,920	576	2,500	77
XMSS$^+$	20	4	16	28,300	124	22	3,744	576	1,956	67
XMSS$^+$	20	4	32	41,500	176	28	3,600	544	1,652	50
XMSS	10	4	4	14,600	86	22	1,680	608	2,292	92
XMSS	10	4	16	18,800	100	17	1,648	576	1,236	78
XMSS	16	4	4	925,400	134	23	2,448	800	2,388	86
XMSS	16	4	16	1,199,100	159	18	2,416	768	1,332	72

We used parameter sets with two heights. A key pair with $h = 16$ allows to generate more than $65,000$, one with $h = 20$ to generate more than one million signatures. Assuming a validity period of one year, this corresponds to seven signatures per day and two signatures per minute, respectively. The runtimes show, that XMSS$^+$ key generation can be done on the smart card in practical time. For all but one used parameter set, the key generation time is below 30 seconds. The times for signature generation and verification are all below 200 ms and 30 ms, respectively. The size of the secret key is around four kilo byte and signatures are around two kilo byte, while the public keys are around 500 bytes. Increasing the tree height for XMSS almost doubles key generation time. For XMSS$^+$ the key generation time is almost doubled if one increases the height by two, as this means that the height of each internal tree is increased by one.

The results show that we can reduce the signature size by increasing the Winternitz parameter w. The behavior of the implementation reflects the theory. The factor for the reduction of the W-OTS signature size is only logarithmic in w. The increase of the runtime is negligible for small w. This can be explained by the following. While the length of the single function chains increases, the number of chains decreases. For $w > 16$ the increase of the runtime becomes almost linear. So from this point, $w = 16$ seems to be a good choice. On the other hand, the provable security level also decreases almost linearly in w. While this only reflects a provable lower bound on the security of the scheme, it is still another reason to keep w small.

Table 2. Results for XMSS$^+$ for message length $m = 256$ on an Infineon SLE78 for different values of k. We use the same k and w for both trees. The table shows the worst case signing times, as well as the average case times.

				Timings (ms)			Size (byte)
Scheme	h	k	w	KeyGen	Sign (w.c.)	Sign (avg.c.)	Secret key
XMSS$^+$	16	0	16	6,700	133	96	3,312
XMSS$^+$	16	2	16	6,700	118	96	3,200
XMSS$^+$	16	4	16	6,700	97	83	3,232
XMSS$^+$	16	6	16	7,000	95	67	4,352
XMSS$^+$	16	8	16	8,000	94	53	10,112

Table 2 shows two things. On the one hand, it is possible to decrease the average case signing time spending more storage for the secret key state, by increasing k. This is what one assumes given the theory. On the other hand, the worst case signing time can only be reduced up to a certain limit. For the given parameters this limit is 94ms, the worst case signing time, when both trees are completely stored. These 94ms are mainly caused by the write operations, when one key pair on the lower level is finished. While all the computations are done in previous rounds, the data structures for the next lower level key pair have to be copied to the data structure for the current lower level key pair. Further the new data structures for the next lower level key pair must be initialized. Choosing $k = 4$ seems to be the most reasonable choice for $h = 16$.

Comparison. The last rows of Table 1 show the results for classical XMSS. The results show that XMSS key generation can be done on the smart card, but is impractical as it already takes more than 15 minutes for $h = 16$. Increasing the height by one almost doubles the runtime of key generation. Generating a key with XMSS$^+$ is already for $h = 16$ almost 200 times faster than with XMSS. While XMSS$^+$ signature generation is slightly faster for comparable parameters, verification is faster for XMSS. The faster key generation is paid by slightly bigger secret keys and signatures, while the XMSS$^+$ public keys are smaller, because of the reused bitmasks.

Now we compare XMSS$^+$ with RSA 2048 and ECDSA 256 on the same smart card. The key generation performance of XMSS$^+$ is similar to RSA 2048, which needs on average 11 seconds, but slower than ECDSA 256 (95ms). Signature generation is comparable to RSA 2048 (190ms) and ECDSA 256 (100ms). Only verification takes slightly longer than with RSA 2048 (7ms), but it is faster than with ECDSA 256 (58ms). The security level of RSA 2048 and ECDSA 256 is 95 and 128 bits, respectively. In contrast to the security level shown in Table 1, these numbers are not based on a security proof, but on the best known attacks. As mentioned above, the security level of XMSS$^+$ is 128 bit, when we only assume the best known attacks.

NVM. The changing key presents a challenge for the implementation of XMSS$^+$ and XMSS on smart cards. NVM is organized in sectors and pages. Due to physical limitations only complete pages can be written (erased and reprogrammed) at once. Furthermore they wear out and cannot be programmed anymore after a certain number of write cycles, depending on the technology (about $500,000$ in our case). However, as write operations are distributed over all 33 physical pages of a sector, the complete available cycles are around 16.5 million per sector.

Generating a key takes only a few hundred write cycles, but its state has to be updated after each signature step. Overall, one million available signatures require one million write cycles for the modification of the state. Using careful memory management, layout and optimization, we managed to keep the number of write cycles below five million for a key pair with $h = 20$, which is far below the 16.5 million available per sector. This includes key generation and all 2^{20} signatures. It should be noted, that this affects only one NVM sector of the card. To use multiple keys, they can be placed in different sectors in order to preserve NVM quality.

5 Conclusion

We presented the first smart card implementation of a forward secure signature scheme. The results presented in Section 4 show that the implementation is practical and that key generation can be done on the card in less than a minute. This is in contrast to previous implementations of similar schemes, that did not achieve on-card key generation. To achieve this, we introduced XMSS$^+$, an improved version of XMSS. Besides the improved key generation, the worst case signing time is also reduced. While the presented improvement is necessary for an implementation on smart cards, it might also show to be useful for implementations on other hardware (At least in cases, where key generation time or worst case signing time are critical).

Given the results of the last section, we propose the parameter set $h = 16$, $w = 16$ and $k = 4$. These parameters seem to lead the optimal performance as long as $65,000$ signatures per key pair are enough. The provable lower bound on the security level of 71 bits is too low from a theoretical point of view. But if we compute the security level according to the best known attacks - as it is common practice - we get a security level of 128 bit. This leads to interesting directions for future work. One would be to either tighten the security proofs or find better reductions from different security assumptions. Another one would be to implement XMSS$^+$ with co-processors for block ciphers with a bigger block size than AES. Alternatively, it would be possible to use hash functions with a digest length of more than 128 bit, using the constructions from [6] to construct the PRF.

One topic we did not address in this work is the side channel resistance. But the forward security property already protects against the most common attack vector for side channel attacks. If a user looses her smart card and revokes her key pair, an attacker can not gain any advantage of a successful side channel attack. The secret key the adversary learns is revoked from this time on and it is not possible to learn the keys of prior time periods. Nevertheless, as there exist other attack vectors, it would be interesting to analyze the side channel resistance of our implementation.

References

1. Abdalla, M., Miner, S.K., Namprempre, C.: Forward-Secure Threshold Signature Schemes. In: Naccache, D. (ed.) CT-RSA 2001. LNCS, vol. 2020, pp. 441–456. Springer, Heidelberg (2001)
2. Abdalla, M., Reyzin, L.: A New Forward-Secure Digital Signature Scheme. In: Okamoto, T. (ed.) ASIACRYPT 2000. LNCS, vol. 1976, pp. 116–129. Springer, Heidelberg (2000)
3. Anderson, R.: Two remarks on public key cryptology. Relevant Material Presented by the author in an Invited Lecture at the 4th ACM Conference on Computer and Communications Security, CCS, pp. 1–4. Citeseer (1997) (manuscript)
4. Bellare, M., Miner, S.K.: A Forward-Secure Digital Signature Scheme. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 431–448. Springer, Heidelberg (1999)
5. Buchmann, J., Dahmen, E., Ereth, S., Hülsing, A., Rückert, M.: On the Security of the Winternitz One-Time Signature Scheme. In: Nitaj, A., Pointcheval, D. (eds.) AFRICACRYPT 2011. LNCS, vol. 6737, pp. 363–378. Springer, Heidelberg (2011)
6. Buchmann, J., Dahmen, E., Hülsing, A.: XMSS - A Practical Forward Secure Signature Scheme Based on Minimal Security Assumptions. In: Yang, B.-Y. (ed.) PQCrypto 2011. LNCS, vol. 7071, pp. 117–129. Springer, Heidelberg (2011)
7. Buchmann, J., Dahmen, E., Klintsevich, E., Okeya, K., Vuillaume, C.: Merkle Signatures with Virtually Unlimited Signature Capacity. In: Katz, J., Yung, M. (eds.) ACNS 2007. LNCS, vol. 4521, pp. 31–45. Springer, Heidelberg (2007)
8. Buchmann, J., Dahmen, E., Schneider, M.: Merkle Tree Traversal Revisited. In: Buchmann, J., Ding, J. (eds.) PQCrypto 2008. LNCS, vol. 5299, pp. 63–78. Springer, Heidelberg (2008)
9. Buchmann, J., García, L.C.C., Dahmen, E., Döring, M., Klintsevich, E.: CMSS – An Improved Merkle Signature Scheme. In: Barua, R., Lange, T. (eds.) INDOCRYPT 2006. LNCS, vol. 4329, pp. 349–363. Springer, Heidelberg (2006)
10. Camenisch, J., Koprowski, M.: Fine-grained forward-secure signature schemes without random oracles. Discrete Applied Mathematics 154(2), 175–188 (2006); Coding and Cryptography
11. Cronin, E., Jamin, S., Malkin, T., McDaniel, P.: On the performance, feasibility, and use of forward-secure signatures. In: Proceedings of the 10th ACM Conference on Computer and Communications Security, CCS 2003, pp. 131–144. ACM, New York (2003)
12. Damgård, I.B.: A Design Principle for Hash Functions. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 416–427. Springer, Heidelberg (1990)

13. ETSI. XML advanced electronic signatures (XAdES). Standard TS 101 903, European Telecommunications Standards Institute (December 2010)
14. ETSI. CMS advanced electronic signatures (CAdES). Standard TS 101 733, European Telecommunications Standards Institute (March 2012)
15. Itkis, G., Reyzin, L.: Forward-Secure Signatures with Optimal Signing and Verifying. In: Kilian, J. (ed.) CRYPTO 2001. LNCS, vol. 2139, pp. 332–354. Springer, Heidelberg (2001)
16. Kozlov, A., Reyzin, L.: Forward-Secure Signatures with Fast Key Update. In: Cimato, S., Galdi, C., Persiano, G. (eds.) SCN 2002. LNCS, vol. 2576, pp. 241–256. Springer, Heidelberg (2003)
17. Krawczyk, H.: Simple forward-secure signatures from any signature scheme. In: Proceedings of the 7th ACM Conference on Computer and Communications Security, CCS 2000, pp. 108–115. ACM, New York (2000)
18. Lenstra, A.K.: Key lengths. Contribution to the Handbook of Information Security (2004)
19. Malkin, T., Micciancio, D., Miner, S.K.: Efficient Generic Forward-Secure Signatures with an Unbounded Number Of Time Periods. In: Knudsen, L.R. (ed.) EUROCRYPT 2002. LNCS, vol. 2332, pp. 400–417. Springer, Heidelberg (2002)
20. Matyas, S., Meyer, C., Oseas, J.: Generating strong one-way functions with cryptographic algorithms. IBM Technical Disclosure Bulletin 27, 5658–5659 (1985)
21. Merkle, R.C.: One Way Hash Functions and DES. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 428–446. Springer, Heidelberg (1990)
22. Rohde, S., Eisenbarth, T., Dahmen, E., Buchmann, J., Paar, C.: Fast Hash-Based Signatures on Constrained Devices. In: Grimaud, G., Standaert, F.-X. (eds.) CARDIS 2008. LNCS, vol. 5189, pp. 104–117. Springer, Heidelberg (2008)
23. Song, D.X.: Practical forward secure group signature schemes. In: Proceedings of the 8th ACM Conference on Computer and Communications Security, CCS 2001, pp. 225–234. ACM, New York (2001)

A Security Level

We compute the security level in the sense of [18]. This allows a comparison of the security of XMSS$^+$ with the security of a symmetric primitive like a block cipher for given security parameters. Following [18], we say that XMSS$^+$ has security level b if a successful attack on the scheme can be expected to require approximately 2^{b-1} evaluations of functions from F_n and $\mathcal{H}_n$. Following the reasoning in [18], we only take into account generic attacks on $\mathcal{H}_n$ and F_n. A lower bound for the security level of XMSS was computed in [6]. For XMSS$^+$, we combined the exact security proofs from [6] and [19]. Following the computation in [6], we can lower bound the security level b by

$$b \geq \max\{n - h/2 - 4 - w_u - 2log(\ell_u w_u), n - h - 4 - w_l - 2log(\ell_l w_l)\}$$

for the used parameter sets.

Extracts from the SHA-3 Competition

Vincent Rijmen*

KU Leuven, Dept. ESAT/SCD-COSIC and IBBT, Belgium

1 The Start of the Competition

The story of the SHA-3 competition starts with the presentation of surprisingly efficient attacks on several modern hash functions at Eurocrypt 2005 [1,2] and at Crypto 2005 [3,4]. Collisions were given for the hash functions MD4, MD5, RIPEMD and SHA-0. An algorithm was shown that can produce collisions for SHA-1 with a complexity that is much lower than previously thought. Before 2005, there were already partial attacks known for several of these hash functions, but only MD4 was really broken [5]. Soon the results were furthere improved and extended to other hash functions. These developments caused NIST to start an effort to develop and standardize a new Secure Hashing Algorithm. This effort was going to be an open competition, similar to the AES competition which it had run from 1998 until 2000.

In 1998, NIST received 21 submissions for the AES competition, of which 15 fulfilled the formal submission requirements and were allowed to enter the competition. In 2008, NIST received 64 submissions for the SHA-3 competition, of which 51 were allowed to enter. This big increase in candidate algorithms and the limited amount of effort available implied that NIST had to adopt a strategy that would allow it to quickly reduce the number of candidates. In July 2009, barely 6 months after the first SHA-3 Candidate Conference, NIST announced the 14 algorithms that were selected to enter the second round of the competition. Quite some of the other 37 candidates had been broken or turned out to be very slow compared to the current standards. However, it can also be said that some of the non-selected algorithms were victims of bad PR. In December 2010, NIST announced the 5 finalist algorithms, which were allowed to enter the third and last round of the competition.

2 The 5 Finalists

We briefly discuss the main elements of each of the 5 finalists (in alphabetical order). The reader will notice that NIST selected the finalists in such a way that all corners of the design space are covered. This is probably not a coincidence.

* This work was sponsored by the Research Fund KU Leuven (OT/08/027) and by the European Commission through the ICT Programme under Contract ICT-2007-216676 (ECRYPT II). The information in this paper is provided as is, and no 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.

Blake [6] is an *Addition-Rotation-XOR (ARX)* design. It has a round function inspired by the stream cipher Salsa20 [7]. Its internal state is represented by a square, which in each round is first processed column by column, like in the AES [8], and subsequently diagonal by diagonal. Blake has 14 or 16 rounds, depending on the desired length of the digest, which determines the size of the internal state. Despite its large number of rounds and the fact that each round contains two passes over the whole state, Blake is very fast on almost all platforms.

Grøstl [9] is a *Substitution-Permutation Network (SPN)* design. Among the finalists, Grøstl comes the closest to the AES and Rijndael designs. It uses a state consisting of two 8×8 squares for 256-bit digests, respectively of two 8×16 rectangular arrays for 512-bit digests. The round transformation uses the AES S-box and diffusion operations which are very similar to the AES diffusion operations. There are 10 or 14 rounds, depending on the digest length. Grøstl is classified as a *permutation based* design, which is distinguished from a *block-cipher based* design by the fact that the message is not inputted in every round, but only at the start and the end of the compression function.

Also JH [10] is a permutation-based SPN design. JH has 42 rounds. It uses 4-bit S-boxes and generalizes the two-dimensional AES diffusion to 8 dimensions, however without generalizing the proof on the minimum number of active S-boxes. Digests of length 256 or 512 bits are produced by exactly the same function.

Keccak [11] is a *Sponge* design, which implies that it is also permutation-based. The Keccak permutation has 24 rounds. It uses a 5-bit S-box and bit-level diffusion with a nice geometric representation. Keccak is a streaming design: every message bit is added to the state only once. The state is a 3-dimensional array. Specifically, it is a $5 \times 5 \times z$ array, where z is equal to 64 for the version submitted to the competition. The versions producing digests of 256 respectively 512 bits differ only in the number of message bits that are added to the state between two applications of the permutation. On most platforms, the speed of Keccak is in the middle of the 5 finalists.

Skein [12] is the second ARX design. It has 72 rounds and is very fast on high-end platforms. Skein is block-cipher based. It uses the block cipher Threefish in the newly designed *Unique Block Iteration (UBI)* mode. Threefish. was designed specifically for Skein and bears no resemblance to Twofish or Blowfish. Digests of length 256 or 512 bits are produced by exactly the same function.

3 Speed

The eBASH site offers speed measurements of the finalists on a multitude of desktop and server platforms, but also a few smaller processors [13]. It indicates that Skein and (the 256-bit version of) Blake are usually the fastest, Keccak is mostly in the middle. At the bottom side, Grøstl-512 is usually the slowest, preceded by JH and Grøstl-256. On platforms with the special AES instructions, Grøstl-256 beats Keccak. Hence, ARX designs appear to make the best use of these platforms.

The XBX study looks at embedded processors and takes the requirements for ROM and RAM into account [14]. In this study, Blake usually performs well, and it never performs badly. On many high-end platforms, Skein is the fastest. Keccak and Grøstl perform repeatedly among the best, in particular when small footprint is important.

4 Highlights

If the talks and papers presented during the AES competition would have to be summarized in a few words, we would propose security-margin weighted performance, side-channel attack resistance, algebra and the pronounciation of certain Dutch vowels. For the SHA-3 competition, these would be replaced by provable security/indifferentiability, rebound, practicality of attacks and distinguishers/nonrandom properties.

The indifferentiability concept [15] predates the SHA-3 conference. It has now become common practice to produce an indifferentiability proof for every new design [16]. Like with all security proofs, a certain level of idealisation is required in order for the proofs to work. At which level the idealization is done, influences for instance the provable security of Blake [17]. Clearly, every methodology which forces designers to reason about the security properties of their design, is a good thing to have. The exact implications of an indifferentiability proof for the security of a design remain a topic of further research [18].

A distinguisher for a keyed primitive is a well-defined concept. For an unkeyed primitive, say a standardized hash function, defining a distinguisher turns out to be problematic. There appears to be a vague intuition in folk lore, that any property of a specific hash function, that is present with small probability only in a *random* function, should be considered as a weakness, since it allows to *distinguish* the specific hash function from a random function. The problem is that distinguishing a deterministic hash function h from a random function f, is trivial. For example, we can simply query the function on 0; the probability that $f(0) = h(0)$ is negligible. More broadly speaking, some of the recently presented results can be considered as *Texas sharpshooter fallacies*, because they don't take into account the effect that is also known as the *law of truly large numbers*: since there is an infinite number of properties that can be investigated, any hash function is bound to exhibit some of them.

5 Outlook

Given the relative slow progress in cryptanalytic results on SHA-256 and the quite competitive speed of SHA-256 and SHA-512 on many processors, it is likely that the winner of the SHA-3 competition will face a hard time pushing its way into applications. For comparison, consider how long it took AES to replace the much slower 3-DES or even the clearly insecure DES in many applications.

On the other hand, it is clear that the competition has increased the pace of progress in knowledge on hash function design and cryptanalysis. Some of the new cryptanalytic insights even led to improved results on AES [19,20]. Furthermore, several important steps have been made in the development of tools and libraries assisting the (semi-)automatic cryptanalysis of hash functions and symmetric-key algorithms [21]. It might well be that these will be the most important outcome of the SHA-3 competition.

References

1. Wang, X., Yu, H.: How to break MD5 and other hash functions. In: [22], pp. 19–35
2. Wang, X., Lai, X., Feng, D., Chen, H., Yu, X.: Cryptanalysis of the hash functions MD4 and RIPEMD. In: [22], pp. 1–18
3. Wang, X., Yin, Y.L., Yu, H.: Finding collisions in the full SHA-1. In: [23], pp. 17–36
4. Wang, X., Yu, H., Yin, Y.L.: Efficient collision search attacks on SHA-0. In: [23], pp. 1–16
5. Dobbertin, H.: Cryptanalysis of MD4. J. Cryptology 11(4), 253–271 (1998)
6. Aumasson, J.P., Henzen, L., Meier, W., Phan, R.C.W.: SHA-3 proposal BLAKE, version 1.3 (December 16, 2010) `http://131002.net/blake/blake.pdf`
7. Bernstein, D.J.: The Salsa20 Family of Stream Ciphers. In: Robshaw, M., Billet, O. (eds.) New Stream Cipher Designs. LNCS, vol. 4986, pp. 84–97. Springer, Heidelberg (2008)
8. National Institute of Standards and Technology (NIST): FIPS-197: Advanced Encryption Standard (2001), `http://www.itl.nist.gov/fipspubs`
9. Gauravaram, P., Knudsen, L.R., Matusiewicz, K., Mendel, F., Rechberger, C., Schläffer, M., Thomsen, S.S.: Grøstl, a SHA-3 candidate (March 2, 2011) `http://www.groestl.info/Groestl.pdf`
10. Wu, H.: The hash function JH (January 16, 2011) `http://www3.ntu.edu.sg/home/wuhj/research/jh/jh_round3.pdf`
11. Bertoni, G., Daemen, J., Peeters, M., Assche, G.V.: The Keccak reference, version 3.0 (January 14, 2011), `http://keccak.noekeon.org/Keccak-reference-3.0.pdf`
12. Ferguson, N., Lucks, S., Schneier, B., Whiting, D., Bellare, M., Kohno, T., Callas, J., Walker, J.: The Skein hash function family, version 1.3 (October 1, 2010) `http://www.skein-hash.info/sites/default/files/skein1.3.pdf`
13. Bernstein, D.J., Lange, T. (eds.): eBACS: ECRYPT benchmarking of cryptographic systems, `http://bench.cr.yp.to/ebash.html`
14. Wenzel-Benner, C., Gräf, J., Pham, J., Kaps, J.P.: XBX benchmarking results (May 2012), `https://xbx.das-labor.org/trac/export/82/page/trunk/documentation/benchmarking_results_may_2012.pdf`
15. Maurer, U.M., Renner, R.S., 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)
16. Andreeva, E., Mennink, B., Preneel, B.: Security Reductions of the Second Round SHA-3 Candidates. In: Burmester, M., Tsudik, G., Magliveras, S., Ilić, I. (eds.) ISC 2010. LNCS, vol. 6531, pp. 39–53. Springer, Heidelberg (2011)
17. Andreeva, E., Luykx, A., Mennink, B.: Provable security of BLAKE with non-ideal compression function. IACR Cryptology ePrint Archive 2011 (2011) 620
18. Ristenpart, T., Shacham, H., Shrimpton, T.: Careful with Composition: Limitations of the Indifferentiability Framework. In: Paterson, K.G. (ed.) EUROCRYPT 2011. LNCS, vol. 6632, pp. 487–506. Springer, Heidelberg (2011)

19. Biryukov, A., Khovratovich, D.: Related-Key Cryptanalysis of the Full AES-192 and AES-256. In: Matsui, M. (ed.) ASIACRYPT 2009. LNCS, vol. 5912, pp. 1–18. Springer, Heidelberg (2009)
20. Bogdanov, A., Khovratovich, D., Rechberger, C.: Biclique Cryptanalysis of the Full AES. In: Lee, D.H. (ed.) ASIACRYPT 2011. LNCS, vol. 7073, pp. 344–371. Springer, Heidelberg (2011)
21. ECRYPT II Symlab: Tools for cryptography, http://www.ecrypt.eu.org/tools/
22. Cramer, R. (ed.): EUROCRYPT 2005. LNCS, vol. 3494. Springer, Heidelberg (2005)
23. Shoup, V. (ed.): CRYPTO 2005. LNCS, vol. 3621. Springer, Heidelberg (2005)

Cryptanalysis of the “Kindle” Cipher

Alex Biryukov, Gaëtan Leurent, and Arnab Roy

University of Luxembourg
{alex.biryukov,gaetan.leurent,arnab.roy}@uni.lu

Abstract. In this paper we study a 128-bit-key cipher called PC1 which is used as part of the DRM system of the Amazon Kindle e-book reader. This is the first academic cryptanalysis of this cipher and it shows that PC1 is a very weak stream cipher, and can be practically broken in a known-plaintext and even in a ciphertext-only scenario.

A hash function based on this cipher has also been proposed and is implemented in the binary editor WinHex. We show that this hash function is also vulnerable to a practical attack, which can produce meaningful collisions or second pre-images.

Keywords: Cryptanalysis, Stream cipher, Hash function, Pukall Cipher, PC1, PSCHF, MobiPocket, Amazon Kindle, E-book.

1 Introduction

In this paper we study the stream cipher PC1, a 128-bit key cipher designed by Pukall in 1991. The cipher was first described in a Usenet post [7] and implementations of the cipher can be found on the designer’s website [9][1]. The PC1 cipher is a part of the DRM system of the MOBI e-book format, which is used in the Amazon Kindle and in MobiPocket (a popular free e-book reader which supports a variety of platforms). This fact makes this cipher into one of the most widely deployed ciphers in the world with millions of users holding devices with this algorithm inside. This cipher is also used in a hashing mode by WinHex [10], an hexadecimal editor used for data recovery and forensics.

So far, no proper security analysis of PC1 is available in the academic literature. Thus it is interesting to study the security of PC1 due to its widespread use and because it offers a nice challenge to cryptanalyst. Our results show practical attacks on the PC1 stream cipher. First, we show a known plaintext attack which recovers the key in a few minutes with a few hundred kilobytes of encrypted text (one small book). Second, we show a ciphertext-only attack, using a secret text encrypted under one thousand different keys. We can recover the plaintext in less than one hour, and we can then use the first attack to extract the keys if needed. Additionally, we show that the hashing mode is extremely weak, by building a simple second-preimage attack with complexity 2^{24}, and a more advanced attack using meaningful messages with a similar complexity. Our results are summarized in Table 1.

[1] Implementations of PC1 can also be found in various DRM removal tools.

2 Description of PC1

PC1 can be described as a self-synchronizing stream cipher, with a feedback from the plaintext to the internal state. The cipher uses 16-bit integers, and simple arithmetic operations: addition, multiplication and bitwise exclusive or (xor). The round function produces one byte of keystream, and the plaintext is encrypted byte by byte.

The internal state of the cipher can be described as a 16-bit integer s, and an 8-bit integer π which is just the xor-sum of all the previous plaintext bytes ($\pi^t = \oplus_{i=0}^{t-1} p^i$). The round function PC1Round takes as input the 128 bit key k and the state (s, π), and will produce one byte of keystream and a new value of the state s.

In this paper we use the following notations:

p	Plaintext	$+$	Addition modulo 2^{16}
c	Ciphertext	$\times$	Multiplication modulo 2^{16}
σ	Keystream	$\oplus$	Boolean exclusive or (xor)
k_i	16-bit sub-keys: $k = k_0 \Vert k_1 \dots \Vert k_7$		
x^t	The value of x after t iterations of PC1Round		
$x[i]$	Bit i of x. We use $x[i–j]$ or $x[i, \dots, j]$ to denote bits i to j.		
$f(x) = x \times 20021 + 1$		$g_i(x) = (x + i) \times 20021$	
$h(x) = x \times 346$		$\mathsf{fold}(x) = (x \gg 8) \oplus x \pmod{2^8}$	

A schematic description of PC1 is shown in Figure 1, and pseudo-code is given in Figure 2. We can divide the PC1 in two parts, as shown in the figure:

– The first part is independent of the state s and takes only the key and the state π as input. We denote this part as KF (key function), and it produces two outputs: $w = \mathsf{KF}_1(\pi, k)$ is a set of 8 values used by the second part, and $\sigma_k = \mathsf{KF}_2(\pi, k)$ is used to create the keystream.
– The second part updates the state s from the previous value of s, and the value of the w’s. We denote this part as SF (state function), and it produces two outputs: $\mathsf{SF}_1(s, w)$ is the new state s, and $\sigma_s = \mathsf{SF}_2(s, w)$ is used to create the keystream.

A high-level representation of PC1 using these functions is given in Figure 3. An important property of PC1 is that the only operations in KF and SF are modular additions, modular multiplications, and bitwise xors (the f, g_i, and h functions only use modular additions and multiplication). Therefore, KF and SF are T-functions [5]: we can compute the i least significant bits of the outputs by knowing only the i least significant bits of the inputs. The only operation that is not a T-function in the PC1 design is the fold from 16 bits to 8 bits at the end.

Note that our description of PC1 does not follow exactly available code: we use an equivalent description in order to make the state more explicit. In particular, we put g_0 at the end of the round in order to only have a 16-bit state s, while the reference code keeps two variables and computes g_0 and the subsequent sum at the beginning of the round. We also use an explicit π state instead of modifying the key in place.

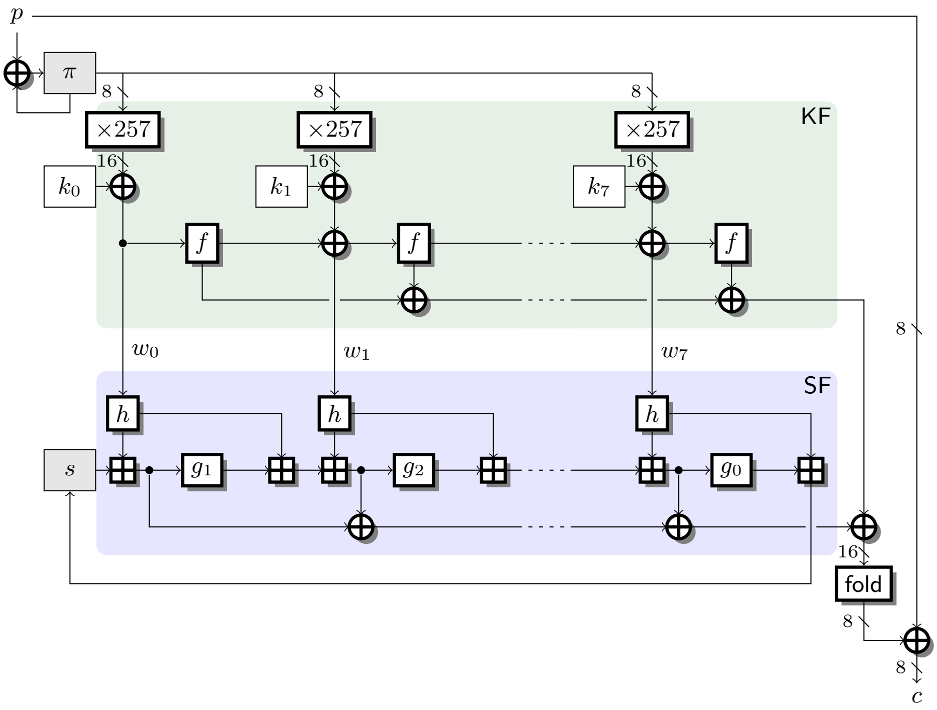


Fig. 1. The PC1 stream cipher

2.1 Use in the Mobipocket e-Book Format and in the Kindle

A notable use of the PC1 stream cipher is in the MOBI e-book format [6]. This format allows optional encryption with PC1; this feature is used to build the DRM scheme of MobiPocket and of the Amazon Kindle. An encrypted e-book is composed of plaintext meta-data and several encrypted text segments. Each segment contains 4 kB of text with HTML-like markup, and is optionally compressed with LZ77. This implementation of LZ77 keeps most characters as-is in the compressed stream, and use non-ASCII characters to encode length-distance pairs. In practice there are still many repetitions in the compressed stream, most of them coming from the formatting tags.

It is a well established fact the DRM system of both MobiPocket and the Amazon Kindle are based on PC1. As a verification, we downloaded several e-books from the Amazon store and they all followed this format.

Each text segment in a given e-book is encrypted with the same key, and the PC1 stream cipher does not use any IV. Thanks to the plaintext feedback the corresponding keystreams will not all be the same. Nonetheless the lack of IV implies a significant weakness: the first byte of keystream will be the same for all encrypted segments, so we can recover the first character of each segment by knowing the first character of the file. Moreover we can detect when two segments share a common prefix.

```
function PC1Round($k$,$\pi$,$s$)
    $k \to k_0, k_1, \ldots, k_7$
    $\sigma \leftarrow 0$
    $w \leftarrow 0$
    for $0 \leq i < 8$ do
        $w \leftarrow w \oplus k_i \oplus (\pi \times 257)$
        $x \leftarrow h(w)$                              ▷ $h(x) = x \times 346$
        $w \leftarrow f(w)$                              ▷ $f(x) = x \times 20021 + 1$
        $s \leftarrow s + x$
        $\sigma \leftarrow \sigma \oplus w \oplus s$
        $s \leftarrow g_{i+1 \bmod 8}(s) + x$            ▷ $g_i(x) = (x + i) \times 20021$
    $\sigma \leftarrow \mathsf{fold}(\sigma)$           ▷ $\mathsf{fold}(x) = (x \gg 8) \oplus x \pmod{2^8}$
    return $(\sigma, s)$

function PC1Encrypt($k$, $p$)
    $\pi \leftarrow 0$;    $s \leftarrow 0$
    for all $p^t$ do
        $(\sigma, s) \leftarrow$ PC1Round$(k, \pi, s)$
        $c^t \leftarrow p^t \oplus \sigma$
        $\pi \leftarrow \pi \oplus p^t$
    return $c$

function PC1Decrypt($k$, $c$)
    $\pi \leftarrow 0$;    $s \leftarrow 0$
    for all $c^t$ do
        $(\sigma, s) \leftarrow$ PC1Round$(k, \pi, s)$
        $p^t \leftarrow c^t \oplus \sigma$
        $\pi \leftarrow \pi \oplus p^t$
    return $p$
```

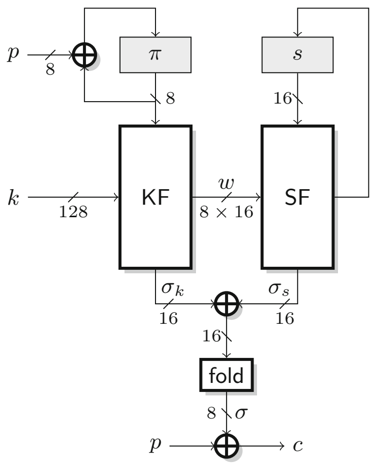
Fig. 2. Pseudo-code of the PC1 stream cipher

Attacks on the DRM Scheme. Like all DRM schemes, this system is bound to fail because the key has to be present in the device and can be extracted by the user. Indeed this DRM scheme has been reverse engineered, and software is available to decrypt the e-books in order to read them with other devices [3].

In this paper, we do not look at the DRM part of the system, but we target the stream cipher from a cryptanalysis point of view.

3 Previous Analysis

Two simple attacks on PC1 have already been described. Our new attacks will exploit some of the same properties — which we rediscovered — and expand on those ideas in order to build practical key-recovery attacks.



$$
\begin{aligned}
\pi^{t+1} &= \pi^t \oplus p^t \\
w^t &= \mathsf{KF}_1(\pi^t, k) \\
s^{t+1} &= \mathsf{SF}_1(s^t, w^t) \\
\sigma^t &= \mathsf{fold}(\mathsf{KF}_2(\pi^t, k) \oplus \mathsf{SF}_2(s^t, w^t)) \\
c^t &= p^t \oplus \sigma^t
\end{aligned}
$$

Fig. 3. Overview of PC1. Grey boxes represent memory registers.

Table 1. Summary of the attacks on the PC1 stream cipher, and on the PSCHF hash function

Attacks on PC1		Complexity	Data	Reference
Distinguisher	Chosen plaintext	2^{16}	2^{16}	[1]
Key recovery	Known plaintext	2^{72}	2^{4}	[2]
Key recovery	Known plaintext	2^{31}	2^{20}	Section 5
Key recovery	Ciphertext only, 2^{10} unrelated keys	2^{35}	$2^{17}.2^{10}$	Section 6
Attacks on PSCHF		**Complexity**		**Reference**
Second preimage	with meaningful messages	2^{24}		Section 7

3.1 Key Guessing

As described above, most of the computations of PC1 can be seen as a T-function. Therefore, if we guess the low 9 bits of each sub-key, we can compute the low 9 bits of KF and SF in a known plaintext attack. This gives one bit of the keystream after the folding, and we can discard wrong guesses. We can then guess the remaining key bits, and the full attack has a complexity of 2^{72}. This was described in a Usenet post by Hellström [2].

3.2 State Collisions

Our description clearly shows that the internal state of the stream cipher is very small: 8 bits in π that do not depend of the key, and 16 bits in s. Therefore we expect that there will be collisions in the state quickly. This was first reported by

Hellström on Usenet in [1], where he described a chosen-plaintext distinguisher: given two messages $x_0 \| y$ and $x_1 \| y$ such that x_0 and x_1 have the same xor sum, the encryption of y will be the same in both messages with probability 2^{-16}.

A more efficient distinguisher can be built using the birthday paradox. We consider 2^8 different prefixes x_i with a fixed xor sum, and a fixed suffix y. When encrypting the messages $x_i \| y$, we expected that two of them will show the same encryption of y when the state s collides after encrypting x_i and x_j.

4 Properties of PC1

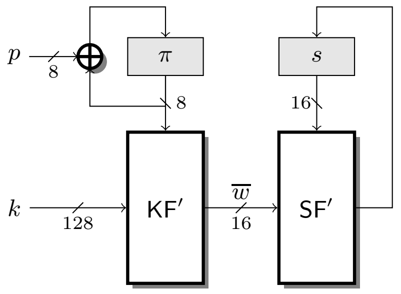
Before describing our attacks, we study some useful properties of the design of PC1.

4.1 Simplified State Update

First, we can see that the SF_1 function only uses modular additions and modular multiplications by constants. Therefore, the state update can be written as a degree 1 polynomial (the full coefficients are given in Appendix A):

$$s^{t+1} = \mathsf{SF}_1(s^t, w^t) = \sum_{i=0}^{7} \left(a_i \times w_i^t\right) + b \times s^t + c$$

If we integrate the computation of $\overline{w} = \sum_{i=0}^{7} a_i \times w_i$ inside KF, we only have to transmit 16 bits between KF and SF for the s update loop. This results in the simplified state update of Figure 4 (we denote the resulting functions by KF' and SF').



$$\begin{aligned} \overline{w}^t &= \mathsf{KF}'(\pi^t, k) \\ s^{t+1} &= \mathsf{SF}'(s^t, \overline{w}^t) \\ &= \overline{w}^t + b \times s^t + c \end{aligned}$$

Fig. 4. Simplified PC1 state update

In the following, we denote $\mathsf{KF}'(x, k)$ as $\overline{w}_x$ for a given key k. In particular, we have $\overline{w}^t = \overline{w}_{\pi^t}$. One can see that knowing the values of $\overline{w}_x$ for all possible x is sufficient to compute the state update without knowing the key itself. Equivalently, we can see KF' as a key-dependent 8×16 bit S-Box.

4.2 Diffusion

All operations in KF and SF are T-function so the only diffusion is from the low bits to the high bits. Moreover, several bits actually cancel out and only affect output bits at higher indexes. We can learn how the key affects the state update by looking at the coefficients a_i. We notice an interesting property of the least significant bits of the coefficients:

$$\forall i, \qquad a_i \equiv 4 \bmod 8 \qquad\qquad b \equiv -1 \bmod 8 \qquad\qquad c \equiv 4 \bmod 8$$

Therefore, if bits 0 to i of the key and plaintext are known, we can compute bits 0 to i of w, and bits 0 to $i+2$ of s. More precisely, we can write

$$\sum_{i=0}^{7} a_i \times w_i = 4 \times \sum_{i=0}^{7} w_i + 8 \times \sum_{i=0}^{7} a'_i \times w_i \qquad\qquad \text{with } a_i = 8a'_i + 4$$

$$s^{t+1} \equiv 4 \times \sum_{i=0}^{7} w_i - s^t + 4 \pmod 8$$

We also have $\sum_{i=0}^{7} w_i \equiv \bigoplus_{i=1,3,5,7} k_i \pmod 2$ and $s^0 = 0$, which leads to:

$$s^t \equiv 4 \times t \times \left(1 \oplus \bigoplus_{i=1,3,5,7} k_i \right) \pmod 8$$

In particular, this shows that $s^t \equiv 0 \bmod 4$ (the two least significant bits of s^t are always zero), and we will always have $s^{t+2} = s^t \bmod 8$. Collisions between s^t and $s^{t'}$ will be more likely if t and t' have the same parity; more generally, collisions are more likely when $t - t'$ is a multiple of large power of two, but the exact relations are difficult to extract.

We can also see that keys with $\bigoplus_{i=1,3,5,7} k_i = 1$ will lead to more frequent collisions, because 3 bits of s are fixed to zero. This defines a class of key keys with regard to collision based attacks. More generally we can define classes of increasingly weak keys for which more low bits of the state are fixed.

5 Collision-Based Known Plaintext Attack

As mentioned above, collisions in the internal state are relatively likely due to the small state size. We show how to use such collisions in an efficient key recovery attack.

First, let us see how we can detect state collisions. If a collision happens between steps t and t' (*i.e.* $s^t = s^{t'}$ and $\pi^t = \pi^{t'}$) this will result in $\sigma^t = \sigma^{t'}$ and $s^{t+1} = s^{t'+1}$. Additionally, if the plaintexts at positions t and t' match, we will have $c^t = c^{t'}$ and $\pi^{t+1} = \pi^{t'+1}$, which will in turn give $\sigma^{t+1} = \sigma^{t'+1}$ and $s^{t+2} = s^{t'+2}$. Furthermore, if several bytes of the plaintext match, this will give a match in several keystream bytes because the state transitions will be the same. More formally, we have

$$\begin{cases} s^t = s^{t'} \\ \pi^t = \pi^{t'} \\ p^{t,\dots,t+u-1} = p^{t',\dots,t'+u-1} \end{cases} \implies \begin{cases} s^{t+1,\dots,t+u+1} = s^{t'+1,\dots,t'+u+1} \\ \sigma^{t+1,\dots,t+u+1} = \sigma^{t'+1,\dots,t'+u+1} \\ c^{t+1,\dots,t+u} = c^{t'+1,\dots,t'+u} \end{cases}$$

A u-byte match in the plaintext results in a u-byte match in the ciphertext, plus one byte in the keystream, provided that the state (s, π) also matches.

In order to exploit this in a key-recovery attack, we look for matches in the plaintext and ciphertext, and we assume that they correspond to state collisions. If we get enough colliding bytes we will have few false positives, and we will learn that $s^t = s^{t'}$ for some values of t and t'. This is very valuable because the computation of s from k is a T-function. We can then recover the key bit by bit: if we guess the least significant bits of k we can compute the least significant bits of s and verify that they collide.

We now study some internal details of the cipher in order to speed-up this key recovery and to make it more practical.

5.1 Detecting State Collisions

We look for pairs of positions (t, t') with:

$$\pi^t = \pi^{t'} \qquad \textbf{and} \qquad p^{t,\dots,t+u-1} = p^{t',\dots,t'+u-1} \tag{A}$$

$$\sigma^{t+1,\dots,t+u+1} = \sigma^{t'+1,\dots,t'+u+1} \tag{B}$$

Condition (A) is a $u+1$-byte condition depending only on the plaintext (we have $\pi^t = \bigoplus_{i=0}^{t-1} p^i$), while condition (B) is a $u+1$-byte condition depending also on the ciphertext. In our attack we use $u = 2$: we have a 3-byte filtering to detect the two-byte event $s^t = s^{t'}$, and we expect few false positives.

However, due to the structure of the cipher, the probability of having (B) is bigger than $2^{-8(u+1)}$, even when $s^t \neq s^{t'}$. First, the most significant bit of s does not affect σ, because its effect is cancelled by the structure of additions and xors. More generally, if we have $s^t \equiv s^{t'} \bmod 2^i$ (*i.e.* an i-bit match in s), then $i+1$ bits of σ will match, and this implies $\Pr(\text{B}) \geq 2^{-(16-i-1)(u+1)}$. For instance, with $u = 2$, we have $\Pr(\text{B}) \geq 2^{-2}$ when $s^t \equiv s^{t'} \bmod 2^{14}$, which would generate many false positives.

In our implementation of the attack, when we detect (B), we only assume that this correspond to $s^t \equiv s^{t'} \bmod 2^{10}$. Experimentally, the probability of detecting (B) with $u = 2$ is around 2^{-15} for random keys and random s states, versus 2^{-24} if $s^t \not\equiv s^{t'} \bmod 2^{10}$. Therefore we have $\Pr\left[s^t \not\equiv s^{t'} \bmod 2^{10} \;\middle|\; (\text{B})\right] \approx 2^{-24}/2^{-15} = 2^{-9}$, and we expect to have no false positives when we use a dozen collisions.

Since we only assume that 10 bits of s are colliding, we can only use these collisions to recover the low 8 bits of the subkeys. For the upper 8 bits of the key we use the output stream σ. If we know $k_j[0\text{–}7]$ and we guess $k_j[8]$, we can compute $\sigma_k[0\text{–}8] \oplus \sigma_s[0\text{–}8]$ before the fold, and one bit of σ after the fold. We can verify our guess by comparing this to the known least significant bit of $c \oplus p$.

Note that most text documents have a relatively low entropy, therefore condition (A) will be satisfied with probability significantly higher than $2^{-8(u+1)}$. In practice, with a sample book of 183 kB (after LZ77 compression), we have 120959 pairs of positions satisfying (A), and for a random encryption key we usually detect between six and one hundred collisions.

5.2 Key Recovery

The basic approach to use those collisions in a key recovery attack is to guess the key bits one by one, and to compute the state in order to exclude wrong guesses.

In a known plaintext attack if we guess $k_j[0\text{–}i]$ for all j then we can compute $w_j[0\text{–}i]$ for all j, and also $s[0, \ldots, i+2]$. We can verify a guess by comparing s^t and $s^{t'}$ up to the bit $i+2$.

However, this essentially requires us to perform a trial encryption of the full text to test each key guess. To build a significantly more efficient attack we consider how s is updated from the key. As explained in Section 4, we have $s^{t+1} = \overline{w}_{\pi^t} + b \times s^t + c$, where the $\overline{w}_x$ can be computed from the key. For a given plaintext p, we can compute π^t at each step, and each s^t can be written as a linear combination of the $\overline{w}_x$:

$$s^t = R^t(\overline{w}_0, \ldots, \overline{w}_{255})$$

with the following relations:

$$R^t = \overline{w}_{\pi^t} + b \times R^{t-1} + c \qquad\qquad R^0 = 0$$

We can compute the coefficients of each R^t from the known plaintext, and every state collision we detect can be translated to an equality $R^t = R^{t'}$. For each guess of the least significant bits of the key, checking those equalities only requires to compute the 256 values $\overline{w}_x$, and to evaluate linear combinations with 256 terms.

Moreover, we can look for sparse relations, so that we don't have to evaluate all the 256 values $\overline{w}_x$. First, note that we also have implicit relations due to the structure of KF': we know that KF is a T-function, and the coefficients a_i used to compute $\overline{w}$ are all multiple of 4; this gives $\overline{w}_x \equiv \overline{w}_y \mod 2^{i+2}$ for all x and y with $x \equiv y \mod 2^i$. We use MAGMA to compute the vector space generated by the collision relations, and we compute the quotient of this space by the implicit relations. The basis of the quotient contains relatively sparse relations, and we find very sparse ones (with only one or two terms) when we restrict the equations to $k[0\text{–}i]$ for a small i, by working in the ring $\mathbf{Z}/2^i\mathbf{Z}$.

Using these relations, a key trial now costs less than 256 evaluations of the round function. In practice this gives a speedup of about 100 times over the staightforward approach.

5.3 Dealing with Independent Message Segments

As mentionned in section 2.1, MOBI e-books are divided into several segments. Each of these segments is encrypted with the same key starting with the initial state $(\pi, s) = (0, 0)$. The segments are too short to find collisions *inside* a given segment, but we can use collisions *between* two different segments just as easily.

Let's assume we detect a collision in the state s, π after enciphering the plaintext p_1 from the initial state and after enciphering the plaintext p_2 from the initial state. We can verify a guess of the least significant bits of the key by enciphering p_1 and p_2 starting from the initial state, and verifying that the least significant bits of the states match.

5.4 Complexity of the Attack

It is difficult to give the precise complexity of the attack because it depends on the number of collisions found, and how much filtering they give. We did some experiments to measure the actual complexity by enciphering a fixed book with random keys, as reported by Figure 5. We found that when we have at least 9 collisions, the median complexity is less than 2^{23} key trial, which take about one minute. The attack is still practical with as low as six collisions, but in this case we have to try around 2^{30} key candidates, which takes a few hours with one core of a typical PC.

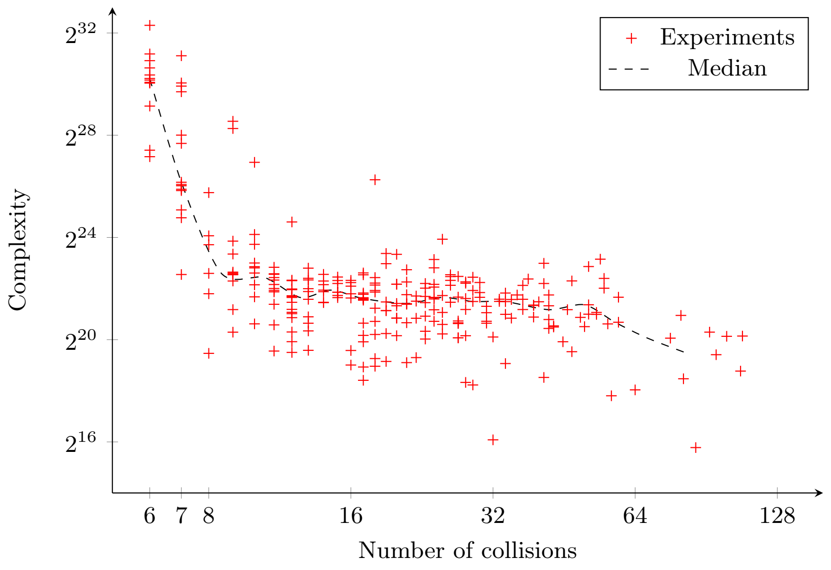


Fig. 5. Experimental results with an e-book of size 336kB (after LZ77 compression). Complexity is shown as the number of key trials.

With this sample e-book, the vast majority of keys result in more than 9 collisions, and often a lot more. This results in a complexity of less that 2^{23} key trials. In general, this will be the case if the known plaintext is sufficiently long (several hundred kilobytes, or a megabyte). Since each key trial costs less than 2^8 evaluation of the round function, the attack will cost less than 2^{31} evaluations of the round functions. In practice, it can be done in less than one minute.

6 Ciphertext Only Attack Using Many Unknown Keys

We now describe a ciphertext-only attack, assuming that the attacker has access to several encryptions of the same text under many different and unrelated keys. If a DRM scheme is based on PC1, this would be a collusion attack where several users buy a copy of a protected work and they share the encrypted data. This attack is based on the observation that the keystream generated by a random key at two differents positions $\sigma_1 = PC1(k, s_1, \pi_1)$ and $\sigma_2 = PC1(k, s_2, \pi_2)$ are biased when $\pi_1 = \pi_2$.

For each plaintext position t, we build a vector C^t with the corresponding ciphertext under each available key. If we consider a pair of positions t and t' the vectors C^t and $C^{t'}$ will be correlated if $\pi^t = \pi^{t'}$. If we manage to detect this correlations efficiently, we can "color" the text positions with 256 colors corresponding to the values of π^t. Then we only have to recover the actual value of π^t corresponding to each color. We can use some known part of the text, the low entropy of the human language, or some extra information recovered when detecting the bias.

Bias in $\boldsymbol{\sigma}$[0]. Let us first study the bias between the C^t vectors. The main bias is in the least significant bit, and is present when $\pi^t[0\text{–}6] = \pi^{t'}[0\text{–}6]$. Let us consider a given plaintext p encrypted under a random key, and two positions t, t' with this property. Because of cancellation effects in the structure of KF and SF, $\pi[7]$ does not affect bits 0–8 of $\sigma_k = \mathsf{KF}_2(\pi, k)$ and $\sigma_l = \mathsf{SF}_2(s, w)$. Moreover, with some probability we have $s^t[0\text{–}8] = s^{t'}[0\text{–}8]$. If this if the case, we will have $\sigma_l^t[0\text{–}8] = \sigma_l^{t'}[0\text{–}8]$, and $\sigma^t[0] = \sigma^{t'}[0]$ after the fold.

This results in a bias in $c^t[0] \oplus c^{t'}[0]$:

$$\begin{aligned}
&\Pr\left[c^t[0] = c^{t'}[0] \;\middle|\; \pi^t[0\text{–}6] = \pi^{t'}[0\text{–}6]\right]\\
&= \Pr\left[p^t[0] \oplus \sigma^t[0] = p^{t'}[0] \oplus \sigma^{t'}[0] \;\middle|\; \pi^t[0\text{–}6] = \pi^{t'}[0\text{–}6]\right]\\
&= \Pr\left[\sigma^t[0] \oplus \sigma^{t'}[0] = p^t[0] \oplus p^{t'}[0] \;\middle|\; \pi^t[0\text{–}6] = \pi^{t'}[0\text{–}6]\right]\\
&\approx 1/2 \pm \Pr\left[s^t[0\text{–}8] = s^{t'}[0\text{–}8]\right]
\end{aligned}$$

The bias is positive if $p^t[0] = p^{t'}[0]$ and negative otherwise. As noted in Section 4.2, two bits of s are fixed to zero, which results in $\Pr\left[s^t[0\text{–}8] = s^{t'}[0\text{–}8]\right] \geq 2^{-7}$. We even have three fixed bits if $t' \equiv t \pmod 2$, which give a stronger bias of 2^{-6}. Moreover, we can get even stronger biases for classes of weak keys, and when using positions such that $t - t'$ is a multiple of a larger power of 2.

Bias with More Bits of σ. There are similar biases with more outputs bits when $\pi^t = \pi^{t'}$. For instance, we have $s^t[0\text{–}9] = s^{t'}[0\text{–}9]$, with some probability. This implies $\sigma_l^t[0\text{–}9] = \sigma_l^{t'}[0\text{–}9]$, and $\sigma^t[0\text{–}1] = \sigma^{t'}[0\text{–}1]$ after the fold. This results in a bias in $c^t[0\text{–}1] \oplus c^{t'}[0\text{–}1]$:

$$\Pr\left[c^t[0\text{–}1] \oplus c^{t'}[0\text{–}1] = p^t[0\text{–}1] \oplus p^{t'}[0\text{–}1] \;\middle|\; \pi^t = \pi^{t'}\right] \approx \frac{1}{4} + \Pr\left[s^t[0\text{–}9] = s^{t'}[0\text{–}9]\right]$$

6.1 Clustering

These biases are quite strong and most colors can be recovered if we have access to a fixed text encrypted under 2^{20} different keys. In order to reduce the number of keys needed, we use a more elaborate algorithm.

First, we work on sets of positions with the same remainder modulo 8: $\mathcal{T}_i = \{t \mid t \equiv i \bmod 8\}$. Positions in the same set will show a stronger bias on $\sigma[0]$: $\Pr\left[s^t[0\text{–}8] = s^{t'}[0\text{–}8] \mid t \equiv t' \pmod 8\right]$ is about 2^{-6} for strong keys, but it can be as high as 2^{-4} for weaker keys (one key in 8 is weak). This allows to detect some relations with only one thousand keys. Each relation also gives the value of $p^t[0] \oplus p^{t'}[0]$ from the sign of the bias.

Then we use a clustering algorithm to detect positions wich share the same color. Initially, we assign a different color to each position, and we merge pairs of colors when we detect a significant bias (we use a priority queue to start with the strongest bias, and we recompute the bias as the clusters grow). When comparing clusters with more than one position, we effectively have a larger sample size, and we can detect weaker biases. Note that we need to correct the signs of the biases using the values of $p^t[0] \oplus p^{t'}[0]$ that we recover when merging colors.

The first phase of the algorithm stops when have identified 128 large colors. We assume that these correspond to the 128 values of $\pi[0\text{–}6]$, which generate the bias in $\sigma[0]$. We then remove false positives from each cluster by verifying that each position is strongly correlated to the rest of the cluster, and we go through all the unassigned positions and assign them to the cluster with the stongest correlation (again, we use a priority queue to start with the strongest bias).

At this point each $\mathcal{T}_i$ has been partitioned in 128 colors, corresponding to the value of $\pi[0\text{–}6]$. We then match the colors of different $\mathcal{T}_i$'s by choosing the strongest correlation (we first merge $\mathcal{T}_i$ and $\mathcal{T}_{i+4}$ because this allows bigger biases, then $\mathcal{T}_i$ and $\mathcal{T}_{i+2}$, and finally $\mathcal{T}_i$ and $\mathcal{T}_{i+1}$).

Finally, we have to split each color: we have the value of $\pi[0\text{–}6]$, but we want to recover the full value of $\pi[0\text{–}7]$. We use the bias on $\sigma[0\text{–}1]$ to detect when two positions correspond to the same $\pi[0\text{–}7]$ (note that we already know the value of $p^t[0] \oplus p^{t'}[0]$). For every color, we first pick two random points to create the new colors, and we assign the remaining points to the closest group. We repeat this with new random choices until the same partition is found three times.

The full attack is given as pseudo-code in Algorithms 1 and 2.

Algorithm 1. Pseudo-code of the clustering algorithm

```
for all $i$ do                                   ▷ Initially, assign a different color to every position
    Color[$i$] $\leftarrow i$

for $0 \le x < 8$ do                                                              ▷ For each $\mathcal{T}_x$
    repeat                                          ▷ Merge colors with the strongest correlation
        for all $i, j \equiv x \pmod 8$, s.t. Color[$i$] $\neq$ Color[$j$] do
            $b \leftarrow$ComputeBias( Get(Color[$i$]), Get(Color[$j$]) )
            if $b > b_{max}$ then
                $b_{max} \leftarrow b$; $c_i \leftarrow$ Color[$i$]; $c_j \leftarrow$ Color[$j$]
        for all $k \in$ Get($c_i$) do
            Color[$k$] $\leftarrow c_j$
    until 128 large colors have been identified                          ▷ Store in MainColor[$x$]
    for all $i \equiv x \pmod 8$ do                                         ▷ Remove false positives
        if ComputeBias( $\{i\}$, Get(Color[$i$]) $\setminus \{i\}$ ) $> \epsilon$ then
            Color[$i$] $\leftarrow$ NewColor()
    for all $i \equiv x \pmod 8$ do                  ▷ Assign remaining points to the closest color
        for $0 \le j < 128$ do
            $b \leftarrow$ComputeBias( $\{i\}$, Get(MainColor[$x$][$j$]) )
            if $b > b_{max}$ then
                $b_{max} \leftarrow b$; $c \leftarrow$ MainColor[$x$][$j$]
        Color[$i$] $\leftarrow c$

Merge(0,4); Merge(1,5); Merge(2,6); Merge(3,7);
Merge(0,2); Merge(1,3); Merge(0,1);                          ▷ Merge colors from different $\mathcal{T}_x$
for $0 \le i < 128$ do                                     ▷ Split colors from $\pi$[0–6] to $\pi$[0–7]
    Split(Get(MainColor[0][$i$]))
return Color
```

6.2 Experiments

We performed experiments with a sample text of a few kilobytes that we encrypted with PC1 under 2^{10} different keys. In this setting, our clustering algorithm can recover the colors in half an hour with a desktop PC. To associate the correct π value to each color, we can use the fact the MOBI format encrypts each chunk independently, and that the first character of a book is always a tag opening character "<". This allows to recover the first byte of each segment, and to identify the colors. In the end, we can decipher the full text with only a few errors. From that point, we can use the known-plaintext attack of Section 5 to recover the keys, and to produce a clean plaintext.

One of the most expensive steps of the attack is to compute the bias between each pair of positions in the plaintext. In our implementation, we use 2^{17} bytes of text, divided in 8 sets $\mathcal{T}_i$ of size 2^{14}. Therefore we have to compute 8×2^{27} biases, and each computation requires 2^{10} bit operations, or 2^5 word operations. Therefore the complexity of the attack is about 2^{35} word operations.

Algorithm 2. Functions used by the clustering algorithm

```
function SPLIT($\mathcal{S}$)                          ▷ Split color from $\pi[0–6]$ to $\pi[0–7]$
    repeat
        $a, b \leftarrow$ RANDOM($\mathcal{S}$); $\mathcal{A} \leftarrow \{a\}$; $\mathcal{B} \leftarrow \{b\}$
        for all $i \in \mathcal{S}$ do
            if COMPUTEBIAS2($\{i\}, \mathcal{A}$) $>$ COMPUTEBIAS2($\{i\}, \mathcal{B}$) then
                $\mathcal{A} \leftarrow \mathcal{A} \cup \{i\}$
            else
                $\mathcal{B} \leftarrow \mathcal{B} \cup \{i\}$
    until the same partition $\mathcal{A}, \mathcal{B}$ is found three times
    $c \leftarrow$ NEWCOLOR()
    for all $i \in \mathcal{A}$ do
        Color[$i$] $\leftarrow c$
function MERGE($x$, $y$)                        ▷ Merge colors from different $\mathcal{T}_x$
    for $0 \le j < 128$ do
        for $0 \le i < 128$ do
            $b \leftarrow$ COMPUTEBIAS( GET(MainColor[$x$][$i$]), GET(MainColor[$y$][$j$]) )
            if $b > b_{max}$ then
                $b_{max} \leftarrow b$; $c \leftarrow$ MainColor[$x$][$i$]
        for all $k \in$ GET(MainColor[$y$][$j$]) do
            Color[$k$] $\leftarrow c$
function GET($c$)                               ▷ Returns the set of positions currently in color $c$
    return $\{i \mid$ Color[$i$] $= c\}$
function COMPUTEBIAS($\mathcal{S}$, $\mathcal{S}'$)          ▷ Evaluates the bias in $\sigma[0]$ between $\mathcal{S}$ and $\mathcal{S}'$
function COMPUTEBIAS2($\mathcal{S}$, $\mathcal{S}'$)         ▷ Evaluates the bias in $\sigma[0–1]$ between $\mathcal{S}$ and $\mathcal{S}'$
```

7 PSCHF: A Hash Function Based on PC1

A hash function based on PC1 has also been proposed by Pukall [8], and it is used in the WinHex hexadecimal editor to check the integrity of a file. The PSCHF hash function operates in two steps:

- First the message is encrypted with PC1 using a fixed key k_h, and the encrypted message is cut into chunks of 256 bits which are xor-ed together to produce an intermediate 256-bit value h.
- Second, a *finalization* function is computed from the final value of the state (s, π), h and the message length $\alpha \pmod{32}$.

The pseudo-code for the hash function is given in Figure 6.

7.1 Second Preimage Attack

To conclude our analysis, we describe a second preimage attack against this hash function. We ignore the finalization, and target the values h, s, π, α after the main loop.

```
$h[0,\dots,31] \leftarrow 0$
$s \leftarrow 0,\ \pi \leftarrow 0$
$\alpha \leftarrow 0$
$\triangleright$ The first loop reads the input message
for all $p^t$ do
    $(\sigma, s) \leftarrow \textsc{PC1Round}(k_h, \pi, s)$
    $h[\alpha] \leftarrow h[\alpha] \oplus \sigma \oplus p^t$
    $\pi \leftarrow \pi \oplus p^t$
    $\alpha \leftarrow \alpha + 1 \bmod 32$
$\triangleright$ The second loop is a finalization whose input are $h$, $s$, $\pi$, and $\alpha$
for $0 \le j < 10 \times (\ell + 1)$ do
    $(\sigma, s) \leftarrow \textsc{PC1Round}(k_h, \pi, s)$
    $\pi \leftarrow \pi \oplus h[\alpha]$
    $h[\alpha] \leftarrow \sigma$
    $\alpha \leftarrow \alpha + 1 \bmod 32$
return $h$
```

Fig. 6. Pseudo-code of the PSCHF hash function. The key k_h is a fixed constant, and ℓ is used to compute the number of blank rounds in the finalization. WinHex uses $\ell = 10$ and $k_h =$ `0xF6C72495179F3F03C6DEF156F82A8538`.

We use $E(M, s, \pi)$ to denote the encryption of a message block M with the key k_h, starting from state (s, π). The intermediate value h can be written as:

$$h = E(M_0, \pi^0, s^0) \oplus E(M_1, \pi^{32}, s^{32}) \oplus \cdots \oplus E(M_l, \pi^t, s^t),$$

where the s^t, π^t values are implicitly computed by the previous E calls. The message blocks are 32-byte long, but the last one might be incomplete.

We can easily build a second preimage attack due to the small internal state of the cipher. We consider a given message $\overline{M}$, and the corresponding target state $\overline{h}, \overline{s}, \overline{\pi}, \overline{\alpha}$ before the finalization function. First, let us assume that the length of $\overline{M}$ is a multiple of 32 bytes, *i.e.* $\overline{\alpha} = 0$. We consider a two-block message $M = M_0, M_1$, and we want to reach the pre-specified value $\overline{h}$:

$$h = E(M_0, \pi^0, s^0) \oplus E(M_1, \pi^{32}, s^{32}) = \overline{h},$$

or equivalently:

$$E(M_1, \pi^{32}, s^{32}) = \overline{h} \oplus E(M_0, \pi^0, s^0).$$

We can find solutions by picking M_0 randomly and just compute M_1 by decrypting $\overline{h} \oplus E(M_0, \pi^0, s^0)$, starting from the state (π^{32}, s^{32}) reached after encrypting M_0. Note that we have $\alpha = 0$ because we use a message of length 64 bytes. However, we also need to reach the pre-specified internal state *i.e.* $s^{64} = \overline{s}, \pi^{64} = \overline{\pi}$. For a random choice of M_0 this should be satisfied with probability 2^{-21} (the probability is higher than 2^{-24} because at least three bits of s are fixed to zero).

If the length of the given message $\overline{M}$ is not a multiple of 32 we can still mount a similar attack. We use a message M made of three parts: M_0 of length α, M_1 of length $32 - \alpha$, and M_2 of length α. A preimage has to satisfy:

$$h = \big(E(M_0, \pi^0, s^0) \| E(M_1, \pi^\alpha, s^\alpha)\big) \oplus \big(E(M_2, \pi^{32}, s^{32}) \| 0^{32-\alpha}\big) = \overline{h}$$
$$\big(E(M_2, \pi^{32}, s^{32}) \| E(M_1, \pi^\alpha, s^\alpha)\big) = \big(E(M_0, \pi^0, s^0) \| 0^{32-\alpha}\big) \oplus \overline{h}$$

Like in the previous case, we can choose a random M_0, and obtain M_1 by decrypting $\overline{h}[\alpha + 1, \dots, 31]$ (starting from the state (π^α, s^α) found after M_0) and M_2 by decrypting $E(M_0, \pi^0, s^0) \oplus \overline{h}[0, \dots, \alpha]$ (starting from the state (π^{32}, s^{32}) found after computing M_1). At the end, we have $\pi^{32+\alpha} = \overline{\pi}$ and $s^{32+\alpha} = \overline{s}$ with probability 2^{-21}.

We can also use the attack with a chosen prefix. Given a target message $\overline{M}$ and a chosen prefix $\overline{N}$, we can build M such that $H(\overline{N} \| M) = H(\overline{M})$.

This attack has been verified and examples of second preimage are given in Table 2. These examples are preimages of the empty message; they can be used as a prefix to any chosen message $\overline{M}$ and will provide a message $P \| \overline{M}$ with the same hash value. In this setting, it is also possible to build a meaningful message: if the message block M_0 is meaningful and goes to the state $s = 0, \pi = 0$, then the decryption of M_0 will give $M_1 = M_0$ and the full message is meaningful.

Table 2. Examples of second preimage of the empty message. We use the same key as in WinHex: k_h = `0xF6C72495179F3F03C6DEF156F82A8538`.

Random Message
D5 06 35 27 03 5C 71 E0 F6 D8 49 9B C9 ED 95 B2
FE 38 1E A0 A5 26 1A 80 91 F8 53 2E EF 5D 54 C4
FC 8B F0 09 D2 5C 5A 36 08 D6 41 F8 34 F5 50 5D
96 F6 C5 30 56 4A 9C 0D E2 DA 29 FD 4C 4A F0 62
Meaningful Message (hex)
2A 20 20 44 4F 20 6E 4F 74 20 52 45 41 44 20 74
68 69 73 20 6D 65 73 73 61 67 65 20 21 20 20 0A
2A 20 20 44 4F 20 6E 4F 74 20 52 45 41 44 20 74
68 69 73 20 6D 65 73 73 61 67 65 20 21 20 20 0A
Meaningful Message (ASCII)
*␣␣DO␣nOt␣READ␣this␣message␣!␣␣
*␣␣DO␣nOt␣READ␣this␣message␣!␣␣
$H(M) = H(\varnothing)$
51 DE 77 DF 24 04 D0 37 18 DE 7C 53 9E 8A 62 75
FA 48 B0 3C E3 C1 5F 31 4D 58 F8 D8 FF 3B 19 8D

7.2 Meaningful Preimages

More generally, we can build arbitrary preimage where we control most of the text, using Joux's multi-collision structure [4], and a linearization technique.

First, we consider 2^8 meaningful blocks with the same xor-sum, and we compute the state after encrypting them. We expect that two block $m_{0,0}$ and $m_{0,1}$ will lead to the same state s^{32}, π^{32}. We repeat this from the state s^{32}, π^{32} to find two messages $m_{1,0}$ and $m_{1,1}$ that lead to the same state s^{64}, π^{64}, and we build a multi-collision structure with 256 pairs $m_{i,0}, m_{i,1}$ iteratively. This structure contains 2^{256} different messages all leading to the same state s^{8192}, π^{8192}. Each step needs 2^8 calls to PC1Round, so the full structure will be built for a cost of 2^{16}.

We then add a final block m_{256} of size $\overline{\alpha}$ and whose xor-sum is $\overline{\pi} \oplus \pi^{8192}$, in order to connect the state s^{8192}, π^{8192} to the target state $\overline{s}, \overline{\pi}$. This require 2^{16} trials. We now have 2^{256} messages all going to the correct $\overline{s}, \overline{\pi}$ and $\overline{\alpha}$, and we will select one that goes to the correct $\overline{h}$ using a linearization technique.

Let us define $c_{i,x} = E(m_{i,x}, \pi^{32i}, s^{32i})$ and $c_{256} = E(m_{256}, \pi^{8192}, s^{8192})$. We can then express h as a function of 256 unknown x_i's:

$$\begin{aligned} h &= c_{0,x_0} \oplus c_{1,x_1} \oplus \cdots \oplus c_{255,x_{255}} \oplus c_{256} \\ &= c_{256} \oplus \bigoplus_{i=0}^{255} c_{i,x_i} \\ &= c_{256} \oplus \bigoplus_{i=0}^{255} c_{i,0} \oplus \bigoplus_{i=0}^{255} x_i \cdot (c_{i,0} \oplus c_{i,1}) \\ &= C \oplus X \cdot D, \end{aligned}$$

where $C = c_{256} \oplus \bigoplus_{i=0}^{255} c_0$, D is a matrix whose row are the $c_{i,0} \oplus c_{i,1}$, and X is a row vector of the x_i's. We can then solve $\overline{h} = C \oplus X \cdot D$ using linear algebra, and we find a message that is a preimage of $\overline{M}$. This technique will produce meaningful second preimages of length around 256 block, *i.e.* 8 kilobytes, with a complexity of 2^{24}.

8 Conclusion

In this work, the study the cipher PC1, which is used in the Amazon Kindle as part of the DRM scheme. Our analysis target the cipher itself, and not the full DRM scheme. We show devastating attacks against the PC1 cipher, and the PSCHF hash function: a known-plaintext key-recovery attack, a ciphertext only attack using a thousand unrelated keys, and a meaningful second-preimage attack on the hash function. All these attacks are practical and have been implemented. While trying to make our attacks more efficient, we have used cryptanalytic techniques which could be of independent interest.

Our attack scenarios are practical: if a DRM scheme is based on PC1, colluding users can recover the plaintext from a thousand ciphertexts encrypted with different keys. This analysis shows that PC1 is very weak and probably made its way into popular products due to the lack of academic cryptanalysis, which we provide in this paper. However, the practical impact on existing DRM schemes is limited, because there are already easy ways to circumvent them.

The main problem in the design of PC1 is the very small internal state, which allows attacks based on internal collisions. Additionnaly, our attacks exploit the fact that several components of PC1 are T-functions, *i.e.* the diffusion is only from the low bits to the high bits.

Acknowledgment. Gaëtan Leurent is supported by the AFR grant PDR-10-022 of the FNR Luxembourg.

References

1. Hellström, H.: Re: Good stream cipher (other than ARCFOUR). Usenet post on sci.crypt (January 18, 2002) Message id: S8K18.14572$l93.3141016@newsb.telia.net
2. Hellström, H.: Re: stream cipher mode. Usenet post on sci.crypt (February 3, 2002) Message id: 3C5CA721.9080905@streamsec.se
3. i♡cabbages: Circumventing Kindle For PC DRM (updated). Blog entry (December 20, 2009) http://i-u2665-cabbages.blogspot.com/2009/12/circumventing-kindle-for-pc-drm.html
4. Joux, A.: Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions. In: Franklin, M. (ed.) CRYPTO 2004. LNCS, vol. 3152, pp. 306–316. Springer, Heidelberg (2004)
5. Klimov, A., Shamir, A.: A New Class of Invertible Mappings. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 470–483. Springer, Heidelberg (2003)
6. MobileRead: MobileRead Wiki — MOBI (2012), http://wiki.mobileread.com/w/index.php?title=MOBI&oldid=30301 (accessed May 14, 2012)
7. Pukall, A.: crypto algorithme PC1. Usenet post on fr.misc.cryptologie (October 3, 1997) Message id: 01bcd098$56267aa0$LocalHost@jeushtlk
8. Pukall, A.: Description of the PSCHF hash function. Usenet post on sci.crypt (June 9, 1997) Message id: 01bc74aa$ae412ae0$1aa54fc2@dmcwnjdz
9. Pukall, A.: The PC1 Encryption Algorithm – Very High Security with 128 or 256-bit keys (2004), http://membres.multimania.fr/pc1/
10. WinHex: WinHex webpage, http://www.x-ways.net/winhex/

A Details of the State Update Polynomial

$$\begin{aligned} s^{t+1} &= \mathsf{SF}_1(s^t, w^t) \\ &= 8460 \times w_0^t - 20900 \times w_1^t - 3988 \times w_2^t - 21444 \times w_3^t + 13004 \times w_4^t \\ &\quad - 20196 \times w_5^t + 16428 \times w_6^t - 19204 \times w_7^t - 15007 \times s^t - 29188 \end{aligned}$$

Cryptographically Strong de Bruijn Sequences with Large Periods

Kalikinkar Mandal and Guang Gong

Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, N2L 3G1, Canada
{kmandal,ggong}@uwaterloo.ca

Abstract. In this paper we first refine *Mykkeltveit et al.*'s technique for producing de Bruijn sequences through compositions. We then conduct an analysis on an approximation of the feedback functions that generate de Bruijn sequences. The cycle structures of the approximated feedback functions and the linear complexity of a sequence produced by an approximated feedback function are determined. Furthermore, we present a compact representation of an $(n + 16)$-stage nonlinear feedback shift register (NLFSR) and a few examples of de Bruijn sequences of period 2^n, $35 \leq n \leq 40$, which are generated by the recursively constructed NLFSR together with the evaluation of their implementation.

Keywords: de Bruijn sequences, nonlinear feedback shift registers, pseudorandom sequence generators, span n sequences, compositions.

1 Introduction

Recently, *nonlinear feedback shift registers* (NLFSRs) have received a lot of attention in designing cryptographic primitives such as Pseudorandom sequence generators (PRSGs) and stream ciphers to provide security and privacy in communication systems. For example, well-known stream ciphers such as Grain and Trivium used NLFSRs as the basic building blocks in their designs [4]. Due to the efficient hardware implementations, NLFSRs have a number of applications in constrained environments for instance RFID tags and sensor networks.

The theory of NLFSRs is not well explored. Most of the known results are collectively reported in Golomb's book [9]. To design a secure cryptographic primitive, such as a key stream generator in a stream cipher, an arbitrary NLFSR cannot be used to generate keystreams with unpredictability, since the randomness properties of a sequence generated by an arbitrary NLFSR are not known. A classical approach to use an NLFSR in a keystream generator is to combine it with a linear feedback shift register (LFSR), where the LFSR guarantees the period of an output keystream. A (binary) *de Bruijn sequence* is a sequence of period 2^n in which each n-bit pattern occurs exactly once in one period of the sequence (this is referred to as the *span n property*). A de Bruijn sequence can be generated by an n-stage NLFSR and it has known randomness properties such as long period, balance, span n property [3, 8, 9].

The linear span or linear complexity of a sequence is defined as the length of the shortest LFSR which generates the sequence. The linear complexity of a de Bruijn sequence is greater than half of its period [2]. However, one can delete one zero bit from the run of zeros of length n of a de Bruijn sequence of period 2^n. The resulting sequence is called a *modified de Bruijn* or *span n sequence*. A span n sequence keeps the balance property and span n properly of the corresponding de Bruijn sequence except for linear span, which could be very low. A classic example of this phenomenon is m-sequences, which are a class of span n sequences that can be generated by an LFSR. By this technique, one can generate a de Bruijn sequence from an m-sequence. The linear complexity of this type of de Bruijn sequences is at least $2^{n-1} + n + 1$ [2]. Likewise, from this de Bruijn sequence, one can remove a zero from the run of zeros of length n then it becomes an m-sequence with linear complexity n. Thus, the lower bound of the linear complexity of this de Bruijn sequence drops to n only after removing one zero from the run of zeros of length n [12]. This shows that the linear complexity of a de Bruijn sequence is not an adequate measurement for its randomness. Instead, it should be measured in terms of the linear complexity of its corresponding span n sequence, since they have only one bit difference.

A de Bruijn sequence and a span n sequence are an one-to-one correspondence, i.e., a span n sequence can be produced from a de Bruijn sequence by removing one zero from the run of zeros of length n. A number of publications in the literature have been discussed several techniques for generating de Bruijn sequences [1, 5–7, 16, 18, 21]. In most of the techniques, a de Bruijn sequence is produced by joining many small cycles, which enforces that either the procedure needs some extra memory for storing the state information for joining the cycles or the feedback function must contain many product terms in order to join the cycles. Most of the existing methods are not efficient for producing de Bruijn sequences of period $2^n, n \geq 30$.

The objective of this paper is to investigate how to generate a de Bruijn sequence where the corresponding span n sequence has a large linear complexity through an iterative method or a composition method. The contribution of this paper is that first we refine *Mykkeltveit et al.*'s iterative method [21] for generating a large period de Bruijn sequence from a feedback function of a short stage feedback shift register which generates a span n sequence. Then we give an analysis of the recursively constructed nonlinear recurrence relation from a cryptographic point of view. In the analysis, we investigate an approximation of the feedback function by setting some product terms as constant functions, and determine the cycle structure of an approximated feedback function and the linear complexity of a sequence generated by an approximated feedback function. The analysis also shows that the de Bruijn sequences generated by the composition have strong cryptographic properties if the starting short span n sequence is strong. Thirdly, we derive a compact representation of an $(n + 16)$-stage NLFSR and present a few instances of cryptographically strong de Bruijn sequences with periods in the range of 2^{35} and 2^{40} together with the discussions of their implementation issues.

The paper is organized as follows. In Sect. 2, we define some notations and recall some background results that are used in this paper. Sect. 3 presents the recursive construction of the arbitrary stage NLFSRs. In Sect. 4, we analyze the feedback functions of the recursive NLFSRs from a cryptographic point of view. In Sect. 5, we present a few instances of cryptographically strong de Bruijn sequences with periods in the range of 2^{35} and 2^{40}. In Sect. 6, we describe the methods for optimizing the number of additions for computing the feedback function of an 40-stage recursive NLFSR. Finally, in Sect. 7, we conclude the paper.

2 Preliminaries

In this section, we define and explain some notations, terms and mathematical functions that will be used in this paper.

- $\mathbb{F}_2 = \{0, 1\}$: the Galois field with two elements.
- $\mathbb{F}_{2^t}$: a finite field with 2^t elements that is defined by a primitive element α with $p(\alpha) = 0$, where $p(x) = c_0 + c_1x + \cdots + c_{t-1}x^{t-1} + x^t$ is a primitive polynomial of degree t (≥ 2) over $\mathbb{F}_2$.
- Z_o^n and Z_e^n denote two sets of odd integers and even integers between 1 and n, respectively.
- $Supp(f)$: the set of all inputs for which $f(x) = 1, x \in \mathbb{F}_{2^n}$, where f is a Boolean function in n variables.
- $H(f)$: the Hamming weight of the Boolean function f.
- $\psi(x_0, x_1) = x_0 + x_1$: a Boolean function in two variables.

2.1 Basic Definitions and Properties

Let $\mathbf{a} = \{a_i\}$ be a periodic binary sequence generated by an n-stage linear or nonlinear feedback shift register, which is defined as [9]

$$a_{n+k} = f(a_k, ..., a_{k+n-1}) = a_k + g(a_{k+1}, ..., a_{k+n-1}),\ a_i \in \mathbb{F}_2,\ k \geq 0 \quad (1)$$

where $(a_0, ..., a_{n-1})$ is called the *initial state* of the feedback shift register, f is a Boolean function in n variables and g is a Boolean function in $(n-1)$ variables. The recurrence relation (1) is called a *nonsingular* recurrence relation. If the function f is an affine function, then the sequence $\mathbf{a}$ is called an *LFSR sequence*; otherwise it is called an *NLFSR sequence*. The *minimal polynomial* of the sequence $\mathbf{a}$ is defined by the LFSR of shortest length that can generate the sequence and the degree of the minimal polynomial determines the linear complexity of the sequence $\mathbf{a}$.

The linear span of a de Bruijn sequence, denoted as LS_{db}, is bounded by $2^{n-1} + n + 1 \leq LS_{db} \leq 2^n - 1$ [2]. On the other hand, the linear span of a span n sequence, denoted as LS_s, is bounded by $2n < LS_s \leq 2^n - 2$ [20]. From this property, we say that a span n sequence has an optimal or suboptimal linear span if its linear span is equal to $2^n - 2$ or close to $2^n - 2$.

It is well known that a nonsingular feedback shift register with a feedback function f partitions the space of 2^n n-tuples into a finite number of cycles, which is known as the *cycle decomposition* or *cycle structure* of f and we denote by $\Omega(f)$ the cycle decomposition of f. Each cycle in $\Omega(f)$ can be considered as a periodic sequence.

Proposition 1. *[9] Let f be a feedback function in n variables that generates a span n sequence, then the function $h = f + \prod_{i=1}^{n-1}(x_i + 1)$ generates a de Bruijn sequence.*

The Welch-Gong (WG) Transformation

Let $\mathrm{Tr}(x) = x + x^2 + \cdots + x^{2^{t-1}}, x \in \mathbb{F}_{2^t}$, be the trace function mapping from $\mathbb{F}_{2^t}$ to $\mathbb{F}_2$. Let $t > 0$ with $(t \bmod 3) \neq 0$ and $3k \equiv 1 \bmod t$ for some integer k. We define a function h from $\mathbb{F}_{2^t}$ to $\mathbb{F}_{2^t}$ by $h(x) = x + x^{q_1} + x^{q_2} + x^{q_3} + x^{q_4}$ and the exponents are given by

$$q_1 = 2^k + 1, q_2 = 2^{2k} + 2^k + 1, q_3 = 2^{2k} - 2^k + 1, q_4 = 2^{2k} + 2^k - 1.$$

The functions, from $\mathbb{F}_{2^t}$ to $\mathbb{F}_2$, defined by

$$f_d(x) = \mathrm{Tr}(h(x^d + 1) + 1) \text{ and } g_d(x) = \mathrm{Tr}(h(x^d)),$$

are known as the *WG transformation* and *five-term (or 5-term) function*, respectively [10, 11], where d is a coset leader which is co-prime with $2^t - 1$. The WG transformation has good cryptographic properties such as high algebraic degree, high nonlinearity. Moreover, a WG sequence has high linear span [11].

2.2 Composite Recurrence Relations

Let $g(x_0, ..., x_{n-1}, x_n) = x_0 + G(x_1, x_2, ..., x_{n-1}) + x_n = 0$ and $f(x_0, ..., x_{m-1}, x_m) = x_0 + F(x_1, x_2, ..., x_{m-1}) + x_m = 0$ be two recurrence relations of n and m stages, respectively that generate periodic sequences, where G and F are Boolean functions in $(n-1)$ and $(m-1)$ variables, respectively. Then, a *composite recurrence relation*, denoted as $g \circ f$, is defined by [21]

$$g \circ f = g(f(x_0, ..., x_m), f(x_1, ..., x_{m+1}), ..., f(x_n, ..., x_{m+n-1})) = 0,$$

which is a recurrence relation of $(n+m)$ stages. The operation "$\circ$" is regarded as the composition operation of recurrence relations. For more detailed treatments on the cycle decomposition of a composite recurrence relation, see [21].

Lemma 1. *[21] Let p be a characteristic polynomial, and $q(x_0, ..., x_n) = x_0 + x_n + w(x_1, ..., x_{n-1})$ where w is a Boolean function in $(n-1)$ variables and let $a \in \Omega(q)$ and $x \in \Omega(q \circ p)$. If the minimal polynomial of a is coprime with p, then $x = b + c$ where b's and a's minimal polynomials are the same and c's minimal polynomial is p.*

Theorem 1. *[21] Let $g = x_0 + x_n + f(x_1, ..., x_{n-1})$, which generates a de Bruijn sequence with period 2^n and let $\psi(x_0, x_1) = x_0 + x_1$. Then both $h_1 = g \circ \psi + \prod_{i \in Z_o^n} x_i \prod_{i \in Z_e^n} (x_i + 1)$ and $h_2 = g \circ \psi + \prod_{i \in Z_o^n} (x_i + 1) \prod_{i \in Z_e^n} x_i$ generate de Bruijn sequences with period 2^{n+1}.*

3 Recursive Feedback Functions in Composed de Bruijn Sequences

In [21], *Mykkeltveit et al.* mentioned the idea of constructing a long stage NLFSR from a short stage NLFSR by repeatedly applying Theorem 1 when g is a linear function in two variables that generates a de Bruijn sequence. In this section, we first refine *Mykkeltveit et al.*'s method and then we show an analytic formulation of a recursive feedback function of an $(n+k)$-stage NLFSR, which is constructed from a feedback function of an n-stage NLFSR by repeatedly applying Theorem 1 and the composition operation.

3.1 The k-th Order Composition of a Boolean Function

Let $g(x_0, x_1, ..., x_n) = x_0 + x_n + G(x_1, x_2, ..., x_{n-1})$ be a Boolean function in $(n+1)$ variables where G is a Boolean function in $(n-1)$ variables. The *first order* composition of ψ and g, denoted as $g \circ \psi$, is given by [21]

$$\begin{aligned} g \circ \psi &= g(x_0 + x_1, x_1 + x_2, ..., x_n + x_{n+1}) \\ &= x_0 + x_1 + x_{n+1} + x_n + G(x_1 + x_2, ..., x_{n-1} + x_n). \end{aligned}$$

Similarly, the *k-th order* composition of g with respect to ψ, denoted as $g \circ \psi^k$, is defined by $g \circ \psi^k = \left(g \circ \psi^{k-1}\right) \circ \psi$, where $g \circ \psi^{k-1}$ is $(k-1)$-th order composition of g with respect to ψ.

3.2 Repeated Compositions of a Product Term

Let X_0^p be a product term in p variables which is given by

$$X_0^p = \prod_{i \in Z_o^p} x_i \prod_{i \in Z_e^p} (x_i + 1).$$

Then the first order composition of X_0^p with respect to ψ, denoted as X_1^p, is given by

$$X_1^p = \prod_{i \in Z_o^p} (x_i + x_{i+1}) \prod_{i \in Z_e^p} (x_i + x_{i+1} + 1)$$

which is a product of sum terms in $(p+1)$ variables. Similarly, the k-th order composition of X_0^p with respect to ψ, denoted by X_k^p, is defined as $X_k^p = (X_{k-1}^p) \circ \psi$, which is a product of sum terms in $(p+k)$ variables. Note that the composition operation with respect to ψ increases the number of variables in X_0^p by one when it repeats once, but the composition operation does not increase the algebraic degree of X_0^p.

We denote by $J^{n-1} = \prod_{i=1}^{n-1}(x_i + 1)$. In a similar manner, the k-th order composition of J^{n-1} with respect to ψ, denoted as J_k^{n-1}, is defined by $J_k^{n-1} = \left(J_{k-1}^{n-1}\right) \circ \psi$, where J_{k-1}^{n-1} is the $(k-1)$-th order composition of J^{n-1}.

Let us now define a function I_k^n in $(n+k-1)$ variables as follows

$$I_k^n(x_1, x_2, ..., x_{n+k-1}) = J_k^{n-1} + X_{k-1}^n + X_{k-2}^{n+1} + \cdots + X_1^{n+k-2} + X_0^{n+k-1}.$$

Then, I_k^n satisfies the following recursive relation

$$I_{k+1}^n = I_k^n \circ \psi + X_0^{n+k}, \text{ for } k \geq 0 \text{ and } n \geq 2,$$

where $I_0^n = J^{n-1}$.

3.3 The Recursive Construction of the NLFSR

In this subsection, we give the construction of an $(n+k)$-stage NLFSR that is constructed from an n-stage NLFSR.

Proposition 2. *Let $g(x_0, x_1, ..., x_n) = x_n + x_0 + G(x_1, x_2, ..., x_{n-1})$, which generates a span n sequence of period $2^n - 1$, where G is a Boolean function in $(n-1)$ variables. Then, for any integer $k \geq 0$, $R_k^n(x_0, x_1, ..., x_{n+k}) = (x_n + x_0) \circ \psi^k + G(x_1, x_2, ..., x_{n-1}) \circ \psi^k + I_k^n(x_1, ..., x_{n+k-1})$ generates a de Bruijn sequence of period 2^{n+k}.*

Proof. By applying Theorem 1 to the feedback function $(g + J^{n-1})$ k times, it becomes

$$\begin{aligned} R_k^n(x_0, x_1, ..., x_{n+k}) &= (x_n + x_0) \circ \psi^k + G(x_1, x_2, ..., x_{n-1}) \circ \psi^k + \\ &\quad I_k^n(x_1, ..., x_{n+k-1}), k \geq 0 \qquad (2) \\ &= (x_n + x_0) \circ \psi^k + G(x_1 \circ \psi^k, ..., x_{n-1} \circ \psi^k) + \\ &\quad I_k^n(x_1, x_2, ..., x_{n+k-1}). \qquad (3) \end{aligned}$$

The function R_k^n is a feedback function in $(n+k)$ variables of an $(n+k)$-stage NLFSR and the recurrence relation, $R_k^n = 0$, generates a de Bruijn sequence with period 2^{n+k}. □

One can construct the feedback function R_{k+1}^n from R_k^n in the following recursive manner

$$R_{k+1}^n = R_k^n \circ \psi + X_0^{n+k} \text{ or } R_{k+1}^n = g \circ \psi^{k+1} + I_{k+1}^n, k \geq 0$$

where $R_0^n = (g + J^{n-1})$.

Remark 1. For $k = 1$, Proposition 2 is the same as Theorem 1 which is also found by Lempel in [18]. For $k = 1$ and g is a primitive polynomial, Proposition 2 is similar to Theorem 2 in [21].

Remark 2. According to Theorem 1, the product term X_0^p in the recurrence relation (2) can be replaced by the product term $\prod_{i\in Z_o^p}(x_i+1)\prod_{i\in Z_e^p} x_i$.

We now present an explicit form of the product terms of I_{16}^n for a recurrence relation of $(n+16)$ stages, which is derived by putting $k=16$ in the recurrence relation (2). Then, the nonlinear recurrence relation of $(n+16)$ stages is given by

$$\begin{aligned} R_{16}^n(x_0,...,x_{n+16}) &= x_{n+16}+x_n+x_0+x_{16}+G(x_1+x_{17},...,x_{n-1}+x_{n+15}) \\ &\quad + J_{16}^{n-1}+X_{15}^n+\cdots+X_1^{n+14}+X_0^{n+15}=0 \end{aligned} \tag{4}$$

where $J_{16}^{n-1}=\prod_{i=1}^{n-1}(x_i+x_{i+16}+1)$ and $X_j^i=T_{o,j}^i\cdot T_{e,j}^i$, $i+j=(n+15)$, $n\le i\le n+15$, $T_{o,j}^i$ and $T_{e,j}^i$ are given in Table 2 (see Appendix). In the product terms, the subscripts o and e represent the odd indices product terms and even indices product terms. Note that each product term X_j^i, $i+j=(n+15)$, $n\le i\le n+15$, is a function of $(n+15)$ variables.

4 Cryptanalysis of the Recursively Constructed NLFSR for Generating de Bruijn Sequences

Since the feedback function contains I_k^n and it includes many product terms whose algebraic degrees are high and the Hamming weights of these product terms are low, as a result, the function I_k^n can be approximated by a linear function or a constant function with high probability. In this section, we first investigate the success probability of approximating the function I_k^n by the zero function. We then study the cycle decomposition of an approximated recurrence relation after a successful approximation of the feedback function with high probability.

4.1 Hamming Weights of the Product Terms and I_k^n

Before calculating the success probability of approximating the function I_k^n by the zero function, we first need to derive the Hamming weight of a composed product term as I_k^n is a sum of $(k+1)$ composed product terms.

Proposition 3. *For an integer $r\ge 1$, the Hamming weight of X_r^p is equal to 2^r.*

Proof. For any product term X_0^p, the r-order composition is of the form $X_r^p=\prod_{i\in Z_o^p} U_i\cdot\prod_{i\in Z_e^p} V_i$, where U_i is a sum of at most $(r+1)$ variables and V_i is also a sum of at most $(r+1)$ variables and the exact number of variables in U_i/V_i depends on the value of r. For simplicity, we assume that $r=2^l, l\ge 0$. To find the Hamming weight of X_r^p, there are two cases to arise.

Case I: When $1 \leq p \leq r+1$
If $r = 2^l$, then U_i and V_j can be written as $U_i = x_i + x_{i+r},\ i \in Z_o^p,\ V_j = (x_j + x_{j+r} + 1),\ j \in Z_e^p$, respectively. $X_r^p = 1$ if and only if $U_i = 1$ and $V_j = 1$ for all $i \in Z_o^p$ and $j \in Z_e^p$. This implies

$$\begin{aligned}
&x_1 = 1 + x_{1+r} = 1 + x_{1+2r} = \cdots = 1 + x_{l_1} = 0/1\\
&x_2 = x_{2+r} = x_{2+2r} = \cdots = x_{l_2} = 0/1\\
&\quad\vdots\\
&x_p = 1 + x_{p+r} = 1 + x_{p+2r} = \cdots = 1 + x_{l_n} = 0/1, \text{ if } p \text{ is odd}\\
&x_p = x_{p+r} = x_{p+2r} = \cdots = x_{l_p} = 0/1, \text{ if } p \text{ is even}
\end{aligned}$$

where $l_i \leq p+r,\ i = 1, 2, ..., p$. Note that X_r^p is a function in $(p+r)$ variables. For an $(p+r)$-tuple with $X_r^p = 1$, the values at $2p$ positions are determined by the values at p positions, which follows from the above set of equations and the remaining $(p+r-2p)$ positions can take any binary value. Hence, the total number of $(p+r)$-tuples for which $X_r^p = 1$ is equal to $2^p \cdot 2^{r-p} = 2^r$.

Case II: When $p \geq r+1$
Similarly, $X_r^p = 1$ if and only if $U_i = 1$ and $V_j = 1$ for all $i \in Z_o^p$ and $j \in Z_e^p$. This implies

$$\begin{aligned}
x_1 &= 1 + x_{1+r} = 1 + x_{1+2r} = \cdots = 1 + x_{l_1} = 0/1\\
x_2 &= x_{2+r} = x_{2+2r} = \cdots = x_{l_2} = 0/1\\
&\ \vdots\\
x_{r-1} &= 1 + x_{2r-1} = \cdots = 1 + x_{l_{r-1}} = 0/1\\
x_r &= x_{2r} = \cdots = x_{l_r} = 0/1
\end{aligned}$$

where $l_i \leq p+r,\ i = 1, 2, ..., r$. According to the above system of equations, the binary values at $(p+r)$ positions are determined by the binary values at r positions and these r positions can take any values. Hence, the total number of $(p+r)$-tuples for which $X_r^p = 1$ is given by 2^r.

By considering $U_i = 1$ and $V_j = 1$ for all $i \in Z_o^p$ and $j \in Z_e^p$ as a system of linear equations with p equations and $(p+r)$ unknown variables over $\mathbb{F}_2$, it follows that the Hamming weight of X_r^p is equal to the number of solutions of the system of linear equations, which is equal to $2^{p+r-r} = 2^r$ for any positive integer $r (\neq 2^l)$. □

Proposition 4. *For any integer $r \geq 1$, the Hamming weight of J_r^{n-1} is equal to 2^r.*

Proof. The proof is similar to the proof of Proposition 3. □

Proposition 5. *For any integer $k \geq 1$ and $n \geq 2$, the Hamming weight of function I_k^n is equal to $2^k + 1$. One can approximate function I_k^n by the zero function with probability $(1 - \frac{1}{2^{n-1}} - \frac{1}{2^{n+k-1}})$.*

Proof. By Proposition 3, the Hamming weight of $X_j^{n+k-1-j}$ is equal to 2^j for $0 \leq j \leq k-1$. Note that $X_j^{n+k-1-j} = 1$ is a system of linear equations with $(n+k-1-j)$ equations and $(n+k-1)$ unknown variables and $Supp(X_j^{n+k-1-j})$ contains the set of all solutions. It is not hard to show that the support of $X_i^{n+k-1-i}$ and $X_j^{n+k-1-j}$ are disjoint for $0 \leq i \neq j \leq n-1$. Again, $(\cup_{j=0}^{k-2} Supp(X_j^{n+k-1-j})) \subset Supp(J_k^{n-1})$, and $Supp(X_{k-1}^{n+k-1})$ and $Supp(J_k^{n-1})$ are disjoint. Then the cardinality of the support of I_k^n is equal to $(2^k + 2^{k-1} - \sum_{j=0}^{k-2} 2^j) = (2^k + 2^{k-1} - 2^{k-1} + 1) = 2^k + 1$. Hence, the Hamming weight of I_k^n is $2^k + 1$.

Since the Hamming weight of I_k^n is 2^k+1, the number of inputs for which I_k^n takes the value zero is equal to $2^{n+k-1} - 2^k - 1$. Hence, one can approximate the function I_k^n by the zero function with probability $(1 - \frac{1}{2^{n-1}} - \frac{1}{2^{n+k-1}})$. □

4.2 Cycle Structures of the Recurrence Relation After Approximation

By Proposition 5, the function I_k^n can be approximated by the *zero function* with probability about $(1 - \frac{1}{2^{n-1}})$. As a consequence, Eq. (2) can be written as follows

$$R_{k,a}^n(x_0, x_1, ..., x_{n+k}) = ((x_n + x_0) + G(x_1, x_2, ..., x_{n-1})) \circ \psi^k. \tag{5}$$

In the following proposition, we provide the cycle structure of the above recurrence relation.

Lemma 2. *For an integer $k \geq 1$, $\Omega(R_{k,a}^n) = \Omega(g) \oplus \Omega(\psi^k)$, i.e., any sequence $x \in \Omega(R_{k,a}^n)$ can be written as $x = b + c$, where b's minimal polynomial is the same as the minimal polynomial of a span n sequence that is generated by g and c's minimal polynomial is $(1+x)^k$ and $\oplus$ denotes the direct sum operation.*

Proof. Let s be a span n sequence generated by g and let $h(x)$ the minimal polynomial of s. Then, $h(x) = h_1(x) \cdot h_2(x) \cdots h_r(x)$, where h_i's are distinct irreducible polynomials of degree less than or equal to n and the value of r depends on the sequence, see [10, 12, 20]. If $h_i(x) = (1+x)$ for some i, then the sequence s is not a span n sequence. On the other hand, the minimal polynomial of ψ^k is $(1+x)^k$. Again, the minimal polynomial of a sequence generated by ψ^k is a factor of $(1+x)^k$. As $h(x)$ does not contain the factor $(1+x)$, the minimal polynomial of s and the minimal polynomial of ψ^k are relatively prime with each other. Then, by Lemma 1, any sequence $x \in \Omega(R_{k,a}^n)$ can be represented by $x = b + c$ where $b \in \Omega(g)$ and $c \in \Omega(\psi^k)$. Hence, the cycle decomposition of $R_{k,a}^n$ is a direct sum of $\Omega(g)$ and $\Omega(\psi^k)$, i.e., $\Omega(R_{k,a}^n) = \Omega(g) \oplus \Omega(\psi^k)$. □

Proposition 6. *The cycle decomposition of $R_{k,a}^n$, i.e., $\Omega(R_{k,a}^n)$ contains $2 \cdot (\Gamma_2(k)+1)$ cycles with $(\Gamma_2(k)+1)$ cycles of period at least $2^n - 1$ and $(\Gamma_2(k)+1)$ cycles of period at most $2^{\lceil \log_2 k \rceil}$, where $\Gamma_2(k)$ is the number of all coset leaders modulo $2^k - 1$.*

Proof. For any positive integer $k \geq 1$, the cycle decomposition of ψ^k is the cycle decomposition of $(1+x)^k$, which contains sequences with period $2^{\lceil \log_2 i \rceil}$, $1 \leq i \leq k$, and the number of cycles is given by $(\Gamma_2(k)+1)$ including the zero cycle (see [9], Th. 3.4, page-42). Again, the cycle decomposition of g contains only two cycles, one is a cycle of length $2^n - 1$ and the other one is the zero cycle of length one. Therefore, by Lemma 2, $\Omega(R^n_{k,a})$ contains $2 \cdot (\Gamma_2(k)+1)$ cycles where $(\Gamma_2(k)+1)$ cycles are of length at least $2^n - 1$ and $(\Gamma_2(k)+1)$ cycles are of length at most $2^{\lceil \log_2 k \rceil}$. □

Proposition 7. *Let $\Omega(R^n_{k,a})$ be the cycle decomposition of $R^n_{k,a}$. For any sequence $x \in \Omega(R^n_{k,a})$ with period at least $2^n - 1$, the linear complexity of x is bounded below by the linear complexity of the sequence generated by g.*

Proof. We already showed in Lemma 2 that any sequence $x \in \Omega(R^n_{k,a})$ can be written as $x = b+c$ where $b \in \Omega(g)$, $c \in \Omega(\psi^k)$, and the minimal polynomial of b is coprime with the minimal polynomial of c. Since the minimal polynomial of b is coprime with the minimal polynomial of c, the linear complexity of x is equal to the sum of the linear complexities of b and c. Therefore, the linear complexity of x is greater than or equal to the linear complexity of b. Hence, the assertion is established. □

Remark 3. Propositions 5, 6, and 7 suggest that in order to generate a strong de Bruijn sequence by this technique, the starting span n sequence generated by g should have excellent randomness properties, particularly, long period and an optimal or suboptimal linear complexity. If an attacker is successful in approximating the feedback function R^n_k by the feedback function $g \circ \psi^k$, then the security of the sequence generated by R^n_k depends on the security of the sequence generated by g.

5 Designing Parameters for Cryptographic de Bruijn Sequences

In this section, we present a few examples of cryptographically strong de Bruijn sequences with period 2^{n+k} for $19 \leq n \leq 24$ and $k = 16$.

5.1 Tradeoff between n and k

From the construction of the recurrence relation, one can determine an $(n+k)$-stage recurrence relation by choosing a small value of n and a large value of k since for a small value of n it is easy to find a span n sequence and the success probability of approximating the feedback function is low. However, for such a choice of the parameters, the recurrence relation contains many product terms, as a result, the function I^n_k may not be calculated efficiently. Thus, for generating a strong de Bruijn efficiently, one needs to choose the parameters in such a way that the nonlinearly generated span n sequence is large enough and its linear complexity is optimal, and the number of product terms in I^n_k is as small as possible.

5.2 Examples of de Bruijn Sequences with Large Periods

Let $\{x_j\}_{j \geq 0}$ be a binary span n sequence generated by the following n-stage recurrence relation for a suitable choice of a decimation number d, a primitive polynomial $p(x)$, and a t-tap position [19]

$$x_n = x_0 + f_d(x_{r_1}, x_{r_2}, ..., x_{r_t}) \tag{6}$$

where $(r_1, r_2, ..., r_t)$ with $0 < r_1 < r_2 < \cdots < r_t < n$ is called a t-tap position and f_d is a WG transformation. Here the decimation number d is a coset leader which is coprime with $2^t - 1$. Then the recurrence relation (4) with G as the WG transformation can be written as

$$\begin{aligned} R_{16}^n = x_{n+16} + x_n + x_0 + x_{16} + f_d(x_{r_1} + x_{r_1+16}, ..., x_{r_t} + x_{r_t+16}) + J_{16}^{n-1} \\ + X_{15}^n + X_{14}^{n+1} + \cdots + X_1^{n+14} + X^{n+15} = 0 \end{aligned} \tag{7}$$

where $J_{16}^{n-1} = \prod_{i=1}^{n-1}(x_i + x_{i+16} + 1)$ and $X_j^p = T_{o,j}^p \cdot T_{e,j}^p, p + j = (n + 15), n \leq p \leq n + 15$, $T_{o,j}^p$ and $T_{e,j}^p$ are given in Table 2. The recurrence relation (7) can generate a de Bruijn sequence for a suitable choice of a decimation number d, a primitive polynomial $p(x)$, and a t-tap position. Our de Bruijn sequences are uniquely represented by the following four parameters: 1) the decimation number d, 2) the primitive polynomial $p(x)$, 3) the t-tap position $(r_1, r_2, ..., r_t)$, and 4) I_k^n.

Table 1 presents a few examples of cryptographically strong de Bruijn sequences with periods in the range of 2^{35} and 2^{40}. In Table 1, the computations for the linear complexity of the 24-stage span n sequence has not finished yet. However, currently the lower bound of the linear complexity is at least 2^{22}. For more instances of span n sequences with an optimal or suboptimal linear span, see [19].

Table 1. De Bruijn sequences with periods $\geq 2^{35}$

WG over $\mathbb{F}_{2^t}$	Decimation	Basis Polynomial	t-tap positions	span n	Linear Span,	I_k^n,	Period
t	d	$(c_0, c_1, ..., c_{t-1})$	$(r_1, r_2, ..., r_t)$	n	span n	k	2^{n+k}
13	55	$(1,1,1,0,1,1,0,0,1,1,0,1,0)$	$(1,2,3,4,5,6,9,10,11,12,13,15,17)$	24	$--$	16	2^{40}
8	53	$(1,1,1,0,0,1,1,1)$	$(1,2,5,6,8,11,12,15)$	21	$2^{21}-5$	16	2^{37}
8	29	$(1,1,1,0,0,0,0,1)$	$(1,2,6,8,9,15,16,19)$	21	$2^{21}-26$	16	2^{37}
8	31	$(1,1,1,0,0,0,0,1)$	$(1,2,10,12,13,16,18,19)$	20	$2^{20}-6$	16	2^{36}
8	1	$(1,1,0,0,0,1,1,0)$	$(1,3,4,5,8,11,12,15)$	19	$2^{19}-2$	16	2^{35}
7	5	$(1,0,0,1,1,1,0)$	$(1,2,6,8,10,12,16)$	20	$2^{20}-7$	16	2^{36}
7	19	$(1,0,1,0,0,1,1)$	$(1,2,3,5,6,10,18)$	19	$2^{19}-2$	16	2^{35}
5	1	$(1,1,1,0,1)$	$(5,10,12,18,19)$	20	$2^{20}-2$	16	2^{36}

Remark 4. In recurrence relation (4), any feedback function g that generates a span n sequence can be used to produce a long de Bruijn sequence. To the best of our knowledge, Table 1 contains a set of (longest) de Bruijn sequences whose algebraic forms of the recurrence relations are known. We here used WG

transformations for producing long period de Bruijn sequences as a span n sequence can be found in a systematic manner by using WG transformations and the compact representation of the recurrence relation (6). In [23], eight span n sequences with periods in the range of $(2^{22} - 1)$ and $(2^{31} - 1)$ are presented and that have been used in stream cipher Achterbahn.

6 Implementation

In this section, we provide some techniques for optimizing the number additions in the product terms for $k = 16$, and give an estimation for the number of multiplications and the time complexity for computing the function I_k^n in terms of n and k.

6.1 Optimizing the Number of Additions

For $k = 16$, I_k^n in recurrence relation (7) contains 17 product terms. For example, for $n = 24$ and $k = 16$, one needs 2116 additions for computing all product terms in I_k^n. In Table 2, we can observe that many partial-sum terms appear in different product terms. By reusing the result of a previously computed sum term, we can optimize the number of additions. For $k = 16$, three optimization rules are described in Table 3.

Applying the rules given in Table 3, the total number of additions required for computing I_{16}^n is given by $(n - 1 + 32 \cdot \lceil \frac{n+5}{2} \rceil + 32 \cdot \lfloor \frac{n+5}{2} \rfloor + 152) = (32 \cdot (n + 5) + n + 151)$, since the numbers of additions required for OR-I, OR-II and OR-III in Table 3 are 32, 18 and 5, respectively. For $n = 24$ and $k = 16$, the number of additions after applying the above three rules is equal to 1103.

6.2 Number of Multiplications and the Time Complexity for Computing I_k^n

The maximum number of multiplications required for computing I_k^n is given by $\sum_{i=n-1}^{n+k-1}(i-1) = (n(k+1) + \frac{(k-1)(k-2)}{2} - 3)$ as one requires $(i-1)$ multiplications to compute a product of i numbers. For $n = 24$ and $k = 16$, the number of multiplications for computing I_k^n equals 510.

Proposition 8. *The time complexity for computing the function I_k^n is approximately given by $\sum_{p=n-1}^{n+k-1} \lceil \log_2 p \rceil$.*

Proof. To compute a product term X_k^p, $n \leq p \leq n + k - 1$, one requires at most $\lceil \log_2 p \rceil$-time. Since the function I_k^n contains $(k + 1)$ product terms, the time complexity for computing I_k^n is given by $\sum_{p=n-1}^{n+k-1} \lceil \log_2 p \rceil$. □

7 Conclusions

In this paper, we first refined a technique by *Mykkeltveit et al.* for producing a long period de Bruijn sequence from a short period span n sequence through the composition operation. We then performed an analysis on the feedback functions of the long period de Bruijn sequences from the cryptographic point of view. In our analysis, we studied an approximation of the feedback functions and the cycle structure of an approximated feedback function, and determined the linear complexity of a sequence generated by an approximated feedback function. In addition, we presented a compact representation of an $(n + 16)$-stage NLFSR and a few instances of de Bruijn sequences with periods in the range of 2^{35} and 2^{40} together with the discussions of their implementation issues. A long period de Bruijn sequence produced by this technique can be used as a building block to design secure lightweight cryptographic primitives such as pseudorandom sequence generators and stream ciphers with desired randomness properties.

References

1. Chan, A.H., Games, R.A.: On the Quadratic Spans of de Bruijn Sequences. IEEE Transactions on Information Theory 36(4), 822–829 (1990)
2. Chan, A.H., Games, R.A., Key, E.L.: On the Complexities of de Bruijn Sequences. Journal of Combinatorial Theory, Series A 33(3), 233–246 (1982)
3. de Bruijn, N.G.: A Combinatorial Problem. Proc. Koninklijke Nederlandse Akademie v. Wetenschappen 49, 758–764 (1946)
4. The eStream Project, http://www.ecrypt.eu.org/stream/
5. Etzion, T., Lempel, A.: Construction of de Bruijn Sequences of Minimal Complexity. IEEE Transactions on Information Theory 30(5), 705–709 (1984)
6. Fredricksen, H.: A Survey of Full Length Nonlinear Shift Register Cycle Algorithms. SIAM Review 24(2), 195–221 (1982)
7. Fredricksen, H.: A Class of Nonlinear de Bruijn Cycles. Journal of Combinatorial Theory, Series A 19(2), 192–199 (1975)
8. Golomb, S.W.: On the Classification of Balanced Binary Sequences of Period 2^n-1. IEEE Transformation on Information Theory 26(6), 730–732 (1980)
9. Golomb, S.W.: Shift Register Sequences. Aegean Park Press, Laguna Hills (1981)
10. Golomb, S.W., Gong, G.: Signal Design for Good Correlation: For Wireless Communication, Cryptography, and Radar. Cambridge University Press, New York (2004)
11. Gong, G., Youssef, A.: Cryptographic Properties of the Welch-Gong Transformation Sequence Generators. IEEE Transactions on Information Theory 48(11), 2837–2846 (2002)
12. Gong, G.: Randomness and Representation of Span n Sequences. In: Golomb, S.W., Gong, G., Helleseth, T., Song, H.-Y. (eds.) SSC 2007. LNCS, vol. 4893, pp. 192–203. Springer, Heidelberg (2007)
13. Good, I.J.: Normal Recurring Decimals. Journal of London Math. Soc. 21(3) (1946)
14. Green, D.H., Dimond, K.R.: Nonlinear Product-Feedback Shift Registers. Proceedings of the Institution of Electrical Engineers 117(4), 681–686 (1970)
15. Green, D.H., Dimond, K.R.: Some Polynomial Compositions of Nonlinear Feedback Shift Registers and their Sequence-Domain Consequences. Proceedings of the Institution of Electrical Engineers 117(9), 1750–1756 (1970)

16. Jansen, C.J.A., Franx, W.G., Boekee, D.E.: An Efficient Algorithm for the Generation of de Bruijn Cycles. IEEE Transactions on Information Theory 37(5), 1475–1478 (1991)
17. Kjeldsen, K.: On the Cycle Structure of a Set of Nonlinear Shift Registers with Symmetric Feedback Functions. Journal Combinatorial Theory Series A 20, 154–169 (1976)
18. Lempel, A.: On a Homomorphism of the de Bruijn Graph and its Applications to the Design of Feedback Shift Registers. IEEE Transactions on Computers C-19(12), 1204–1209 (1970)
19. Mandal, K., Gong, G.: Probabilistic Generation of Good Span n Sequences from Nonlinear Feedback Shift Registers. CACR Technical Report (2012)
20. Mayhew, G.L., Golomb, S.W.: Characterizations of Generators for Modified de Bruijn Sequences. Advanced Applied Mathematics 13(4), 454–461 (1992)
21. Mykkeltveit, J., Siu, M.-K., Tong, P.: On the Cycle Structure of Some Nonlinear Shift Register Sequences. Information and Control 43(2), 202–215 (1979)
22. Rachwalik, T., Szmidt, J., Wicik, R., Zablocki, J.: Generation of Nonlinear Feedback Shift Registers with Special-Purpose Hardware. Report 2012/314, Cryptology ePrint Archive (2012), http://eprint.iacr.org/
23. http://www.ecrypt.eu.org/stream/ciphers/achterbahn/achterbahn.pdf

A. Explicit Forms of Product Terms of I_{16}^n and Optimization Rules

We here present the explicit forms of the product terms of I_{16}^n in Table 2, the rules for optimizing the number of additions required for computing the function I_{16}^n in Table 3, and the product terms of Table 2 after applying the optimization rules in Table 4.

Table 2. Product terms in I_{16}^n of recurrence relation (4)

$T_{o,15}^{n} = \prod_{i\in Z_o^n}\left(\sum_{l=0}^{15} x_{i+l}\right)$	$T_{o,14}^{n+1} = \prod_{i\in Z_o^{n+1}}\left(\sum_{l=0}^{7} x_{i+2l}\right)$
$T_{o,13}^{n+2} = \prod_{i\in Z_o^{n+2}}(x_i + x_{i+1} + \sum_{l=1}^{3}(x_{i+2^l} + x_{i+2^l+1}))$	$T_{o,12}^{n+3} = \prod_{i\in Z_o^{n+3}}(\sum_{l=0}^{3} x_{i+4l})$
$T_{o,11}^{n+4} = \prod_{i\in Z_o^{n+4}}(\sum_{l=0}^{4} x_{i+l} + \sum_{l=8}^{11} x_{i+l})$	$T_{o,10}^{n+5} = \prod_{i\in Z_o^{n+5}}(x_i + x_{i+2} + x_{i+8} + x_{i+10})$
$T_{o,9}^{n+6} = \prod_{i\in Z_o^{n+6}}(x_i + x_{i+1} + x_{i+8} + x_{i+9})$	$T_{o,8}^{n+7} = \prod_{i\in Z_o^{n+7}}(x_i + x_{i+8})$
$T_{o,7}^{n+8} = \prod_{i\in Z_o^{n+8}}(\sum_{l=0}^{7} x_{i+l})$	$T_{o,6}^{n+9} = \prod_{i\in Z_o^{n+9}}(\sum_{l=0}^{3} x_{i+2l})$
$T_{o,5}^{n+10} = \prod_{i\in Z_o^{n+10}}(x_i + x_{i+1} + x_{i+4} + x_{i+5})$	$T_{o,4}^{n+11} = \prod_{i\in Z_o^{n+11}}(x_i + x_{i+4})$
$T_{o,3}^{n+12} = \prod_{i\in Z_o^{n+12}}(\sum_{l=0}^{3} x_{i+l})$	$T_{o,2}^{n+13} = \prod_{i\in Z_o^{n+13}}(x_i + x_{i+2})$
$T_{o,1}^{n+14} = \prod_{i\in Z_o^{n+14}}(x_i + x_{i+1})$	$T_{o,0}^{n+15} = \prod_{i\in Z_o^{n+16}} x_i$
$T_{e,15}^{n} = \prod_{i\in Z_e^n}(\sum_{l=0}^{15} x_{i+l} + 1)$	$T_{e,14}^{n+1} = \prod_{i\in Z_e^{n+1}}(\sum_{l=0}^{7} x_{i+2l} + 1)$
$T_{e,13}^{n+2} = \prod_{i\in Z_e^{n+2}}(x_i + x_{i+1} + \sum_{l=1}^{3}(x_{i+2^l} + x_{i+2^l+1}) + 1)$	$T_{e,12}^{n+3} = \prod_{i\in Z_e^{n+3}}(\sum_{l=0}^{3} x_{i+4l} + 1)$
$T_{e,11}^{n+4} = \prod_{i\in Z_e^{n+4}}(\sum_{l=0}^{4} x_{i+l} + \sum_{l=8}^{11} x_{i+l} + 1)$	$T_{e,10}^{n+5} = \prod_{i\in Z_e^{n+5}}(x_i + x_{i+2} + x_{i+8} + x_{i+10} + 1)$
$T_{e,9}^{n+6} = \prod_{i\in Z_e^{n+6}}(x_i + x_{i+1} + x_{i+8} + x_{i+9} + 1)$	$T_{e,8}^{n+7} = \prod_{i\in Z_e^{n+7}}(x_i + x_{i+8} + 1)$
$T_{e,7}^{n+8} = \prod_{i\in Z_e^{n+8}}(\sum_{l=0}^{7} x_{i+l} + 1)$	$T_{e,6}^{n+9} = \prod_{i\in Z_e^{n+9}}(\sum_{l=0}^{3} x_{i+2l} + 1)$
$T_{e,5}^{n+10} = \prod_{i\in Z_e^{n+10}}(x_i + x_{i+1} + x_{i+4} + x_{i+5} + 1)$	$T_{e,4}^{n+11} = \prod_{i\in Z_e^{n+11}}(x_i + x_{i+4} + 1)$
$T_{e,3}^{n+12} = \prod_{i\in Z_e^{n+12}}(\sum_{l=0}^{3} x_{i+l} + 1)$	$T_{e,2}^{n+13} = \prod_{i\in Z_e^{n+13}}(x_i + x_{i+2} + 1)$
$T_{e,1}^{n+14} = \prod_{i\in Z_e^{n+14}}(x_i + x_{i+1} + 1)$	$T_{e,0}^{n+15} = \prod_{i\in Z_e^{n+16}}(x_i + 1)$

Table 3. Optimization rules for addition

Optimization Rule I			
$Y^1_{1,i} = x_i + x_{i+1}$	$Y^2_{1,i} = x_{i+2} + x_{i+3}$	$Y^1_{3,i} = x_{i+8} + x_{i+9}$	$Y^2_{3,i} = x_{i+10} + x_{i+11}$
$Y^1_{2,i} = x_{i+4} + x_{i+5}$	$Y^2_{2,i} = x_{i+6} + x_{i+7}$	$Y^1_{4,i} = x_{i+12} + x_{i+13}$	$Y^2_{4,i} = x_{i+14} + x_{i+15}$
$Y_{1,i} = Y^1_{1,i} + Y^2_{1,i}$	$Y_{2,i} = Y^1_{2,i} + Y^2_{2,i}$	$Y_{0,2,i} = x_i + x_{i+2}$	$Y_{4,6,i} = x_{i+4} + x_{i+6}$
$Y_{3,i} = Y^1_{3,i} + Y^2_{3,i}$	$Y_{4,i} = Y^1_{4,i} + Y^2_{4,i}$	$Y_{8,10,i} = x_{i+8} + x_{i+10}$	$Y_{12,14,i} = x_{i+12} + x_{i+14}$
$Q_{0,i} = x_i$	$Q_{4,i} = x_i + x_{i+4}$	$Q_{3,i} = Y_{1,i}$	$Q_{7,i} = Q_{3,i} + Y_{2,i}$
$Q_{8,i} = x_i + x_{i+8}$	$Q_{12,i} = Q_{4,i} + x_{i+8} + x_{i+12}$	$Q_{11,i} = Q_{3,i} + Y_{3,i}$	$Q_{15,i} = Q_{7,i} + Y_{3,i} + Y_{4,i}$
$Q_{2,i} = Y_{0,2,i}$	$Q_{6,i} = Q_{2,i} + Y_{4,6,i}$	$Q_{1,i} = Y^1_{1,i}$	$Q_{5,i} = Q_{1,i} + Y^1_{2,i}$
$Q_{10,i} = Q_{2,i} + Y_{8,10,i}$	$Q_{14,i} = Q_{6,i} + Y_{8,10,i} + Y_{12,14,i}$	$Q_{9,i} = Q_{1,i} + Y^1_{3,i}$	$Q_{13,i} = Q_{5,i} + Y^1_{3,i} + Y^1_{4,i}$
Optimization Rule II			
$Y^1_{1,i} = x_i + x_{i+1}$	$Y^2_{1,i} = x_{i+2} + x_{i+3}$	$Y_{1,i} = Y^1_{1,i} + Y^2_{1,i}$	$Y_{2,i} = Y^1_{2,i} + Y^2_{2,i}$
$Y^1_{2,i} = x_{i+4} + x_{i+5}$	$Y^2_{2,i} = x_{i+6} + x_{i+7}$	$Y_i = Y_{1,i} + Y_{2,i}$	$Y_{0,2,i} = x_i + x_{i+2}$
$Y_{4,6,i} = x_{i+4} + x_{i+6}$		$Y_{8,10,i} = x_{i+8} + x_{i+10}$	
$W_{0,i} = x_i$	$W_{1,i} = Y^1_{1,i}$	$W_{4,i} = x_i + x_{i+4}$	$W_{5,i} = Y^1_{1,i} + Y^1_{2,i}$
$W_{2,i} = Y_{0,2,i}$	$W_{3,i} = Y_{1,i}$	$W_{6,i} = Y_{0,2,i} + Y_{4,6,i}$	$W_{7,i} = Y_{1,i} + Y_{2,i}$
$W_{8,i} = x_i + x_{i+8}$	$W_{9,i} = Y^1_{1,i} + x_{i+8} + x_{i+9}$	$W_{10,i} = Y_{0,2,i} + Y_{8,10,i}$	
Optimization Rule III			
$Y_{1,i} = x_i + x_{i+1}$	$Y_{2,i} = x_{i+2} + x_{i+3}$	$Z_{0,1} = x_i$	$Z_{1,i} = Y_{1,i}$
$Z_{2,i} = x_i + x_{i+2}$	$Z_{3,i} = Y_{1,i} + Y_{2,i}$	$Z_{4,i} = x_i + x_{i+4}$	

Table 4. Product terms of recurrence relation ([7])

$T^n_{o,15} = \prod_{i \in Z^n_o} Q_{15,i}$	$T^{n+1}_{o,14} = \prod_{i \in Z^{n+1}_o} Q_{14,i}$
$T^{n+2}_{o,13} = \prod_{i \in Z^{n+2}_o} Q_{13,i}$	$T^{n+3}_{o,12} = \prod_{i \in Z^{n+3}_o} Q_{12,i}$
$T^{n+4}_{o,11} = \prod_{i \in Z^{n+4}_o} Q_{11,i}$	$T^{n+5}_{o,10} = \prod_{i \in Z^{n+5}_o} Q_{10,i}$
$T^{n+6}_{o,9} = \prod_{i \in Z^{n+5}_o} Q_{9,i} \cdot W_{9,n+6}$	$T^{n+7}_{o,8} = \prod_{i \in Z^{n+5}_o} Q_{11,i} \prod^{n+7}_{i=n+6,odd} W_{8,i}$
$T^{n+8}_{o,7} = \prod_{i \in Z^{n+5}_o} Q_{7,i} \cdot \prod^{n+8}_{i=n+6,odd} W_{7,i}$	$T^{n+9}_{o,6} = \prod_{i \in Z^{n+5}_o} Q_{6,i} \cdot \prod^{n+9}_{i=n+6,odd} W_{6,i}$
$T^{n+10}_{o,5} = \prod_{i \in Z^{n+5}_o} Q_{5,i} \cdot \prod^{n+10}_{i=n+6,odd} W_{5,i}$	$T^{n+11}_{o,4} = \prod_{i \in Z^{n+5}_o} Q_{4,i} \cdot \prod^{n+11}_{i=n+6,odd} W_{4,i}$
$T^{n+12}_{o,3} = \prod_{i \in Z^{n+5}_o} Q_{3,i} \cdot \prod^{n+11}_{i=n+6,odd} W_{3,i} \cdot Z_{3,n+12}$	$T^{n+13}_{o,2} = \prod_{i \in Z^{n+5}_o} Q_{2,i} \cdot \prod^{n+11}_{i=n+6,odd} W_{2,i} \cdot \prod^{n+13}_{i=n+12,odd} Z_{2,i}$
$T^{n+14}_{o,1} = \prod_{i \in Z^{n+5}_o} Q_{1,i} \cdot \prod^{n+11}_{i=n+6,odd} W_{1,i} \cdot \prod^{n+13}_{i=n+12,odd} Z_{1,i} \cdot (x_{n+14} + x_{n+15})$	$T^{n+15}_{o,0} = \prod_{i \in Z^{n+16}_o} x_i$
$T^n_{e,15} = \prod_{i \in Z^n_e} Q_{15,i}$	$T^{n+1}_{e,14} = \prod_{i \in Z^{n+1}_e} Q_{14,i}$
$T^{n+2}_{e,13} = \prod_{i \in Z^{n+2}_e} Q_{13,i}$	$T^{n+3}_{e,12} = \prod_{i \in Z^{n+3}_e} Q_{12,i}$
$T^{n+4}_{e,11} = \prod_{i \in Z^{n+4}_e} Q_{11,i}$	$T^{n+5}_{e,10} = \prod_{i \in Z^{n+5}_e} Q_{10,i}$
$T^{n+6}_{e,9} = \prod_{i \in Z^{n+5}_e} Q_{9,i} \cdot W_{9,n+6}$	$T^{n+7}_{e,8} = \prod_{i \in Z^{n+5}_e} Q_{11,i} \prod^{n+7}_{i=n+6,even} W_{8,i}$
$T^{n+8}_{e,7} = \prod_{i \in Z^{n+5}_e} Q_{7,i} \cdot \prod^{n+8}_{i=n+6,even} W_{7,i}$	$T^{n+9}_{e,6} = \prod_{i \in Z^{n+5}_e} Q_{6,i} \cdot \prod^{n+9}_{i=n+6,even} W_{6,i}$
$T^{n+10}_{e,5} = \prod_{i \in Z^{n+5}_e} Q_{5,i} \cdot \prod^{n+10}_{i=n+6,even} W_{5,i}$	$T^{n+11}_{e,4} = \prod_{i \in Z^{n+5}_e} Q_{4,i} \cdot \prod^{n+11}_{i=n+6,even} W_{4,i}$
$T^{n+12}_{e,3} = \prod_{i \in Z^{n+5}_e} Q_{3,i} \cdot \prod^{n+11}_{i=n+6,even} W_{3,i} \cdot Z_{3,n+12}$	$T^{n+13}_{e,2} = \prod_{i \in Z^{n+5}_e} Q_{2,i} \cdot \prod^{n+11}_{i=n+6,odd} W_{2,i} \cdot \prod^{n+13}_{i=n+12,even} Z_{2,i}$
$T^{n+14}_{e,1} = \prod_{i \in Z^{n+5}_e} Q_{1,i} \cdot \prod^{n+11}_{i=n+6,even} W_{1,i} \cdot \prod^{n+13}_{i=n+12,even} Z_{1,i} \cdot (x_{n+14} + x_{n+15})$	$T^{n+15}_{e,0} = \prod_{i \in Z^{n+16}_e} x_i$

Cryptanalysis of the Loiss Stream Cipher

Alex Biryukov[1], Aleksandar Kircanski[2], and Amr M. Youssef[2]

[1] University of Luxembourg
Laboratory of Algorithmics, Cryptology and Security (LACS)
Rue Richard Coudenhove-Kalergi 6, Luxembourg, Luxembourg
[2] Concordia University
Concordia Institute for Information Systems Engineering (CIISE)
Montreal, Quebec, H3G 1M8, Canada

Abstract. Loiss is a byte-oriented stream cipher designed by Dengguo Feng *et al.* Its design builds upon the design of the SNOW family of ciphers. The algorithm consists of a linear feedback shift register (LFSR) and a non-linear finite state machine (FSM). Loiss utilizes a structure called Byte-Oriented Mixer with Memory (BOMM) in its filter generator, which aims to improve resistance against algebraic attacks, linear distinguishing attacks and fast correlation attacks. In this paper, by exploiting some differential properties of the BOMM structure during the cipher initialization phase, we provide an attack of a practical complexity on Loiss in the related-key model. As confirmed by our experimental results, our attack recovers 92 bits of the 128-bit key in less than one hour on a PC with 3 GHz Intel Pentium 4 processor. The possibility of extending the attack to a resynchronization attack in a single-key model is discussed. We also show that Loiss is not resistant to slide attacks.

1 Introduction

Several word-oriented LFSR-based stream ciphers have been recently proposed and standardized. Examples include ZUC [1], proposed for use in the 4G mobile networks and also SNOW 3G [3], which is deployed in the 3GPP networks. The usual word-oriented LFSR-based design consists of a linear part, which produces sequences with good statistical properties and a finite state machine which provides non-linearity for the state transition function.

In 2011, the Loiss stream cipher [4] was proposed by a team from the State Key Laboratory of Information Security in China. The cipher follows the above mentioned design approach: it includes a byte-oriented LFSR and an FSM. The novelty in the design of Loiss is that its FSM includes a structure called a Byte Oriented-Mixer with Memory (BOMM) which is a 16 byte array adopted from the idea of the RC4 inner state. The BOMM structure is updated in a pseudorandom manner.

The Loiss key scheduling algorithm utilizes a usual approach to provide non-linearity over all the inner state bits. During the initialization phase, the FSM output is connected to the LFSR update function. This ensures that after the initialization process, the LFSR content depends non-linearly on the key and the IV. Such an approach has been previously used in several LFSR-based word-oriented constructions such as the SNOW family of ciphers [3]. In Loiss, however, the FSM contains the BOMM element which is updated slowly in a pseudo-random manner. The feedback to the LFSR, used in the initialization phase, passes through this BOMM which turns out to be exploitable in a differential-style attack since the BOMM does not properly diffuse differences.

In this paper, we provide a related-key attack of a practical complexity against the Loiss stream cipher by exploiting this weakness in its key scheduling algorithm (see also [7] for a work that was done independently of our results). The attack requires two related keys differing in one byte, a computational work of around 2^{26} Loiss initializations, $2^{25.8}$ chosen-IVs for both of the related keys, offline precomputation of around 2^{26} Loiss initializations and a storage space of 2^{32} words. This shows that the additional design complication, i.e., the addition of the BOMM mechanism, weakens the cipher instead of strengthening it. We also discuss the possibility of extending such a related-key attack into a resynchronization single-key attack. Finally, we show that Loiss does not properly resist to slide attacks.

The rest of the paper is organized as follows. In section 2, we briefly review relevant specifications of the Loiss stream cipher. Our related-key attack is detailed in section 3 where we also discuss the possibility of extending the attack to the single-key scenario. In section 4, we show that Loiss is not resistant to slide attacks. Finally, our conclusion is given in section 5.

2 Specifications of Loiss

Figure 1 shows a schematic description of the Loiss stream cipher. In here, we briefly review relevant components of the cipher. Let F_{2^8} denote the quotient field $F_2[x]/(\pi(x))$, where the corresponding primitive polynomial $\pi(x) = x^8 + x^7 + x^5 + x^3 + 1$. If α is a root of the polynomial $\pi(x)$ in F_{2^8}, then the LFSR of Loiss is defined over F_{2^8} using the characteristic polynomial

$$f(x) = x^{32} + x^{29} + \alpha x^{24} + \alpha^{-1}x^{17} + x^{15} + x^{11} + \alpha x^5 + x^2 + \alpha^{-1}.$$

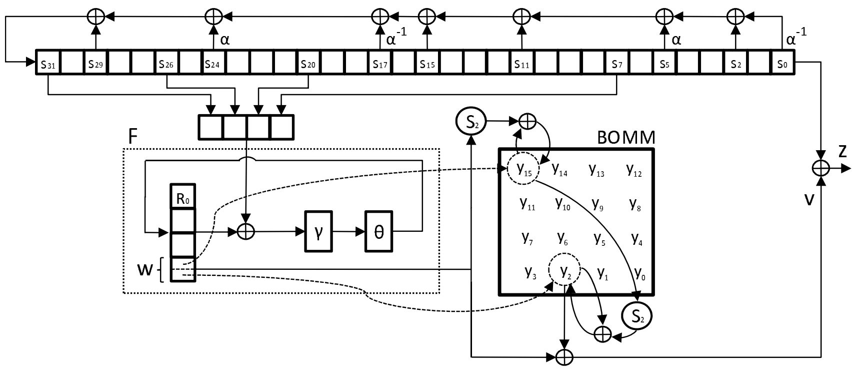


Fig. 1. Loiss stream cipher

The usual bijection between the elements of F_{2^8} and 8-bit binary values is used. The LFSR consists of 32 byte registers denoted by s_i, $0 \leq i \leq 31$. Restating the above equation, if $s_0^t, \ldots, s_{31}^t$ denote the LFSR registers after t LFSR clocks, then the LFSR update function is defined by

$$s_{31}^{t+1} = s_{29}^t \oplus \alpha s_{24}^t \oplus \alpha^{-1} s_{17}^t \oplus s_{15}^t \oplus s_{11}^t \oplus \alpha s_5^t \oplus s_2^t \oplus \alpha^{-1} s_0^t \qquad (1)$$

and $s_i^{t+1} = s_{i+1}^t$ for $0 \leq i \leq 30$.

The FSM consists of the function F and the BOMM. The function F compresses 32-bit words into 8-bit values. It utilizes a 32-bit memory unit R and takes LFSR registers s_{31}, s_{26}, s_{20} and s_7 as input. In particular, in each step, the output of F is taken to be the 8 leftmost bits of the register R, after which the R value is updated by

$$\begin{aligned} X &= s_{31}^t | s_{26}^t | s_{20}^t | s_7^t \\ R^{t+1} &= \theta(\gamma(X \oplus R^t)) \end{aligned}$$

where γ is the S-box layer which uses 8×8 S-box S_1 and is defined by

$$\gamma(x_1|x_2|x_3|x_4) = S_1(x_1)|S_1(x_2)|S_1(x_3)|S_1(x_4)$$

and θ is a linear transformation layer defined by

$$\theta(x) = x \oplus (x \lll 2) \oplus (x \lll 10) \oplus (x \lll 18) \oplus (x \lll 24)$$

Since the attack technique provided in this paper does not depend on the particular choice of the used S-boxes, we refer the reader to [4] for the specifications of S_1 and S_2.

As for the BOMM structure, it utilizes 16 memory units, i.e., bytes $y_0, \ldots, y_{15}$. The BOMM function maps 8-bit values to 8-bit values. Let w and v denote the input and output of the BOMM function. Denote the nibbles of its input w as $h = w \gg 4$ and $l = w \bmod 16$. Then, the BOMM function returns $v = y_h^t \oplus w$, after which the update of its memory units takes place as follows:

```
$y_l^{t+1} = y_l^t \oplus S_2(w)$
If $h \neq l$, then
    $y_h^{t+1} = y_h^t \oplus S_2(y_l^{t+1})$
else
    $y_h^{t+1} = y_l^{t+1} \oplus S_2(y_l^{t+1})$
$y_i^{t+1} = y_i^t$, for $0 \leq i \leq 15$ and $i \notin \{h, l\}$
```

where S_2 is an 8×8 S-box. In the FSM update step, the input to the BOMM function, i.e., the w value, is taken to be leftmost byte of the output of the F function.

The initialization procedure of Loiss proceeds as follows. The register R is set to zero, i.e., $R^0 = 0$. If the key K and the initialization vector IV are represented byte-wise as

$$K = K_{15}|K_{14}|\cdots|K_0$$
$$IV = IV_{15}|IV_{14}|\cdots|IV_0, \tag{2}$$

then the starting inner state $(s_{31}^0, \ldots, s_0^0, R^0, y_{15}^0, \ldots, y_0^0)$ is loaded with the K and IV as follows:

$$s_i^0 = K_i, \;\; s_{i+16}^0 = K_i \oplus IV_i, \;\; y_i^0 = IV_i \tag{3}$$

for $0 \leq i \leq 15$. Then, Loiss runs for 64 steps and the output of the BOMM takes part in the LFSR update step. In other words, instead of (1), the following LFSR update function is used:

$$s_{31}^{t+1} = s_{29}^t \oplus \alpha s_{24}^t \oplus \alpha^{-1} s_{17}^t \oplus s_{15}^t \oplus s_{11}^t \oplus \alpha s_5^t \oplus s_2^t \oplus \alpha^{-1} s_0^t \oplus \boldsymbol{v}^{\mathbf{t}} \tag{4}$$

Then, the keystream generation stage starts. Loiss generator produces one byte of keystream per step:

$$z^t = s_0^t \oplus v^t.$$

In general, except for the new BOMM component, the whole Loiss design is very similar to the design of the SNOW 3G cipher. It is also interesting to note that the same θ linear layer has been used in the SMS4 block cipher [2] and also in ZUC [1].

3 Proposed Attack

In this section, a differential-style attack against the Loiss key scheduling algorithm is presented. The attack requires two related keys that differ in one byte. It also requires the ability to resynchronize the cipher under the two keys with chosen IV values.

The attack starts by having the pair of inner states right after the key loading step differ only in one LFSR byte and one BOMM byte. Then, the idea is to have the LFSR difference fully cancelled. We use the fact that the BOMM output participates in the LFSR update step during the initialization and the BOMM difference helps us to cancel out the LFSR difference through the feedback. Once the difference in the LFSR is fully cancelled, only the BOMM component is active and moreover, with a single byte difference. Then, since the BOMM does not have proper diffusion properties, the single-byte difference stays localized in the BOMM until the end of the initialization, which can be detected from the keystream.

The probability of the event that a given BOMM byte is not used during the initialization is $(\frac{15}{16})^{128} \approx 2^{-12}$, since a BOMM element is consulted 128 times during the 64 initialization steps. If the active byte has not been used until the end of the initialization, the two instances of the cipher generate several equal keystream bytes with high probability. Namely, the difference at the point where the keystream is to be produced will be of low-weight and localized in the BOMM. Therefore, spotting large number of zero bytes in the starting keystream byte difference indicates that the LFSR difference cancellations described above took place. These cancellations happen only when certain equations in the starting LFSR bytes are satisfied and consequently, since the starting LFSR bits are related to the key bits, information about the key bits leaks.

Let K and K' differ only in the byte K_3. The steps of the attack can be summarized as follows:

- Construct a list of (IV, IV') pairs for which the LFSR state difference cancellation happens. The cancellation event is described in section 3.1, the distinguisher used to detect this event is given in section 3.2 and a procedure for collecting the (IV, IV') pairs is provided in section 3.3.
- Use this collection of IVs as input to the filtering procedure to filter the wrong key candidates, as described in section 3.4.

The attack recovers 92 bits of the key and the remaining $128 - 92 = 36$ bits can be obtained by brute force. In another variant of the attack, 112 bits of the key are recovered and the rest are found by brute-force.

3.1 Cancelling the LFSR Difference

In this section, a necessary and sufficient condition for the starting inner state difference to be fully cancelled in the LFSR after 4 steps is provided. The condition is specified in terms of the leftmost byte of the R register in the first 4 steps. Then, the conditions on the R register as provided by Observation 1 below leak information on the early LFSR bytes and thus about the secret key.

The key-loading mechanism (3) allows having a chosen difference only at bytes s_3 and y_3 at time $t = 0$. Namely, it suffices to have

$$K_3 \oplus K_3' = IV_3 \oplus IV_3' = \delta \tag{5}$$

and the rest of the K, K' and also IV, IV' bytes to have a zero-difference. Moreover, the key-loading mechanism trivially allows choosing the starting values of the y_3 register. This is done by choosing the IV_3 byte, since the IV is simply copied into the BOMM. This shows that the assumptions required by Observation 1 (i.e., the particular difference value $0x02$ in s_3 and y_3 and also the $y_3 = 0x9d$ constant) can be satisfied. Recall that w^t denotes the leftmost byte of the R register at time $t \geq 0$.

Observation 1. *Let a pair of Loiss inner states have only s_3 and y_3 bytes active, both with difference $0x02$. Also, let $y_3 = 0x9d$. Then, after 4 steps, the LFSR does not contain any active byte if and only if*

$$(w^0, w^1, w^2, w^3) = (0x00, 0x33, 0xK?, 0x3?) \tag{6}$$

where K is any hexadecimal digit different from $0x3$ and the symbol "?" denotes any hexadecimal digit.

The proof of the observation above is given in Appendix B. Here, a descriptive overview of the cancellation specified by Observation 1 is provided. In Figure 2, the BOMM and the LFSR bytes s_3, s_2, s_1, s_0 are shown during the first four steps. In the second and the fourth states in the figure, the cancellation of the LFSR difference by the feedback byte to the LFSR update is denoted. In the first step, the difference does not enter neither the LFSR update function nor the feedback value (since $w^0 = 0x00$). In the second step, it is required that $w^1 = 0x33$ for the difference to be cancelled and also to be updated to the next necessary BOMM difference value, $2\alpha^{-1}$. In the third step, the difference is neither passed to the LFSR nor changed in the BOMM. Finally, in the fourth step, the difference in the LFSR byte s_0 is cancelled and the LFSR becomes fully inactive.

It should be noted that Observation 1 holds for other difference values apart from $\delta = 0x02$. The set Δ of such differences is given in Appendix A. In particular, Observation 1 is true for any $\delta \in F_2^8$ such that the input differences $\alpha^{-1} \times \delta$ and δ cannot be mapped to the output differences $\alpha^{-1} \times \delta$ by the S_2 S-box (see the ($\Rightarrow$) part of the proof in Appendix B). For each difference from the set Δ, the initial constant for y_0^3 is calculated from (14).

The overall probability that there will be only one BOMM byte, y_3, active after all of the 64 steps of the key scheduling procedure is estimated next. For this event to happen, it suffices to have (6) satisfied in addition to ensuring that the y_3 difference does not propagate to other bytes during the initialization procedure. The event (6) happens with probability $p_w = 2^{-8} \times \frac{15}{16} \times 2^{-4} \approx 2^{-12.1}$. The event by which the y_3 difference does not propagate to any other byte is equivalent to the event of $w^t \bmod 16 \neq 0x3$ and $w^t \gg 4 \neq 0x3$ for $4 \leq t \leq 63$, and $w^2 \bmod 16 \neq 3$. The latter condition is included since Observation 1 does not rule out the possibility of the spreading of the y_3 difference to another byte during step 3. Thus, the probability that y_3 does not spread to any other byte is $p_s = (\frac{15}{16})^{2\times 60+1} \approx 2^{-11.3}$. Thus, a randomly chosen key-IV pair satisfying (5) such that the assumptions of Observation 1 are satisfied produces a pair of inner states with only one active byte with probability

$$p = p_s \times p_w = 2^{-12.1} \times 2^{-11.3} = 2^{-23.4} \tag{7}$$

under the usual independence assumption.

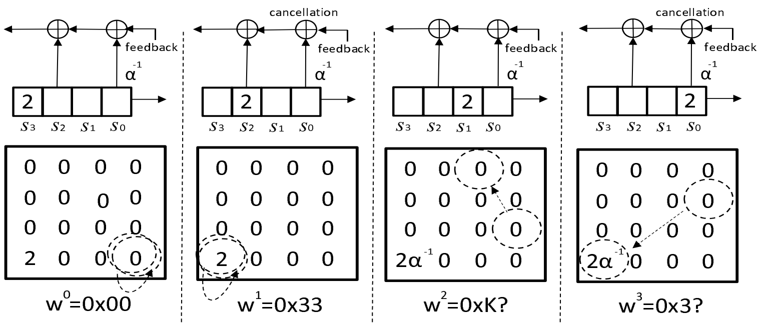


Fig. 2. Illustration of the differences in the BOMM structure at times $t = 0, 1, 2, 3$

3.2 Distinguishing Loiss Pairs

In the previous subsection, we showed that it is possible to have a pair of Loiss inner states with only one active byte (located in the BOMM) after the initialization. Here, a distinguisher for the keystreams generated by a pair of such states is provided. The goal is to minimize the probability of false positives and false negatives.

Let the time at which the two instances of the cipher differ by only one BOMM byte be $t = 0$. Since at this time most of the words are inactive, it is natural to attempt distinguishing Loiss key stream pairs from random keystream pairs by simply counting the number of equal bytes in the two outputs. Such a distinguisher depends on parameters n and m, where n is the number of keystream generation steps that will be considered and m is the number of equal corresponding words in steps $0, \ldots, n-1$. The distinguisher can be formulated as:

- Count the number of indices $0 \leq i < n$ such that $z_i = z'_i$
- If this count is $\geq m$ return *Loiss keystreams*, otherwise return *Random.*

Good values for (n, m) can be chosen by consulting Table 1 Appendix C. In this table, the probability of false positives and false negatives for some representative (n, m) points has been tabulated. Details on how the values in the table have been calculated are provided below.

The false positive probability signifies the probability that in two random sequences of n bytes, more than m corresponding bytes will be equal. On the other hand, the false negative probability signifies the probability that two Loiss instances with only one active byte located in the BOMM, will produce strictly less than m equal bytes. For the purpose of the attack above, it is necessary to keep the probability of false positives low, since a false positive would lead to generating equations that have incorrect key values as solutions.

As for the false positive probability, it has been calculated by using the formula describing the probability that in n randomly generated bytes, at least m of them are equal to zero. Namely, if l denotes the number of zeros in the sample, then $P[\text{false positive}] = P[l \geq m] = \sum_{l=m,\ldots,n} \binom{n}{l} \left(\frac{1}{256}\right)^{l} \left(\frac{255}{256}\right)^{n-l}$.

The false negative probability has been calculated experimentally by randomly generating a pair of equal Loiss inner states and then inducing a random difference at a random BOMM byte. After running the cipher for n steps, the number of equal bytes is counted. If such number is strictly smaller than m, a counter is incremented. After repeating the previous procedure for 2^{28} times and dividing the resulting counter by 2^{28}, an approximation of the probability of a false negatives is obtained.

For the purpose of the distinguisher used in the next subsection, taking $(n, m) = (32, 10)$ makes the probability of the attack failure marginally small, i.e., equal to around $2^{25.8} \times 2^{-54.2}$, since the distinguisher is applied for around $2^{25.8}$ times and a false positive answer would lead to wrong conclusions about the value of key bytes.

3.3 Finding the Correct IVs

According to the cancellation probability (7), for around one in $2^{23.4}$ randomly chosen IVs, if the key-IV pair satisfies (5), the inner state right after the initialization will have only the y_3 BOMM byte active. Given the choices for the distinguisher given in Table 1, such event can be reliably detected. Hereafter, such IVs will be called *correct* IVs. In this section, it is shown that the correct IVs can be found with probability better than $2^{-23.4}$, which helps us reduce the final number of chosen-IVs required for the attack.

In particular, once one correct IV is obtained, more such correct IVs can be found with better probability. Namely, changing certain IV bytes in a correct IV does not influence all w_1, w_2 and w_3 bytes. For instance, perturbing byte IV_{11} in a correct IV does not change $w^1 = 0x33$ value and the the probability (7) that the new IV will also be a correct one increases by a factor of 2^8. More precisely, let T_1 denote a collection of IV bytes such that any change in bytes from T_1 leaves R^1 unchanged, but changes R^t, $t \geq 2$. It is easy to verify that $T_1 = \{IV_1, IV_5, IV_8, IV_{11}, IV_{13}\}$.

Thus, after finding one correct IV, varying only the bytes from T_1 can serve to find more correct IVs with better probability. Such a set of IVs would result in the IVs for which the R^1 word is constant. However, the attack step provided in subsection 3.4, which takes the correct IV set as its input, requires that the IVs produce about 5 different R^1 values. Similarly, there have to be around 360 different R^2 values. These two numbers of required different R^1 and R^2 values are necessary to minimize the number of key byte candidates that will be recovered, as will be explained in the next subsection. Therefore, the search procedure that produces the input to the procedure in the next subsection can proceed as follows:

- Let sets $L_0 = L_1 = L_2 = L_3 = L_4 = \emptyset$.
- Generate 5 correct IVs randomly and place them in sets L_i, $0 \leq i \leq 4$, respectively. In more detail, for each randomly generated IV, compute IV' according to (5) and apply the distinguisher from subsection 3.2. If the distinguisher returns a positive answer, a correct IV has been found.
- For $0 \leq i \leq 4$
  - Using the IV from each L_i, generate more corrects IVs such that the L_i sets contain 72 IVs each. In particular, the new correct IVs are generated by randomizing the starting IV bytes specified by T_1 and applying the distinguisher.

The output of the above procedure are sets L_i, $0 \leq i \leq 3$, each containing 72 IVs for which the R^1 is constant. This procedure takes around $5 \times 2^{23.4} + 5 \times 72 \times 2^{23.4-8} \approx 2^{26}$ chosen-IV queries on both Loiss instances. If instead of applying the previous procedure, all of the $5 \times 72 = 360$ correct IVs were generated randomly, the number of chosen IV queries would be $360 \times 2^{23.4} \approx 2^{31.9}$.

3.4 Filtering the Key Bytes

In each Loiss step, the function F updates the register R by a transformation similar to one round of a block cipher, where the R value plays the role of the plaintext and the four LFSR registers play the role of the round key. The goal hereafter is to recover the LFSR registers fed to F in the first three initialization steps, i.e., s_{7+i}, s_{20+i}, s_{26+i}, s_{31+i} for $0 \leq i \leq 2$. In particular, since the LFSR bytes in question can be represented as a sum of the key and the IV, the goal is to recover the key part in these bytes. First, the application of the F function in the first three steps is represented in the form of

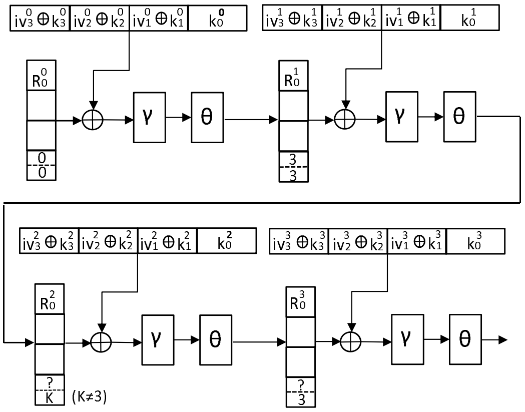


Fig. 3. The R register in times $0 \leq t \leq 3$

$$R^{i+1} = F(R^i, k_3^i \oplus iv_3^i | k_2^i \oplus iv_2^i | k_1^i \oplus iv_1^i | k_0^i) \tag{8}$$

for $0 \leq i \leq 2$, where k_3^i, k_2^i, k_1^i and k_0^i depend only on the original key bytes and iv_3^i, iv_2^i and iv_1^i depend only on the IV bytes. More precisely, in the first step

$$\begin{aligned} k_3^0 &= K_{15}, k_2^0 = K_{10}, k_1^0 = K_4, k_0^0 = K_7 \\ iv_3^0 &= IV_{15}, iv_2^0 = IV_{10}, iv_1^0 = IV_4 \end{aligned} \tag{9}$$

In the second step, we have

$$\begin{aligned} k_3^1 &= K_{13} \oplus \alpha K_8 \oplus \alpha^{-1} K_1 \oplus K_{15} \oplus K_{11} \oplus \alpha K_5 \oplus K_2 \oplus \alpha^{-1} K_0 \\ k_2^1 &= K_{11}, k_1^1 = K_5, k_0^1 = K_8 \\ iv_3^1 &= IV_{13} \oplus \alpha IV_8 \oplus \alpha^{-1} IV_1 \oplus IV_{15} \oplus IV_{11} \oplus \alpha IV_5 \oplus \\ &\qquad\qquad IV_2 \oplus \alpha^{-1} IV_0 \oplus f^1 \\ iv_2^1 &= IV_{11}, iv_1^1 = IV_5 \end{aligned} \tag{10}$$

and in the third step

$$\begin{aligned}
&k_3^2 = K_{14} \oplus \alpha K_9 \oplus \alpha^{-1} K_2 \oplus K_0 \oplus K_{12} \oplus \alpha K_6 \oplus K_3 \oplus \alpha^{-1} K_1 \\
&k_2^2 = K_{12}, k_1^2 = K_6, k_0^2 = K_9 \\
&iv_3^2 = IV_{14} \oplus \alpha IV_9 \oplus \alpha^{-1} IV_2 \oplus IV_0 \oplus IV_{12} \oplus \alpha IV_6 \oplus IV_3 \oplus \\
&\qquad\qquad \alpha^{-1} IV_1 \oplus f^2 \\
&iv_2^2 = IV_{12}, iv_1^2 = IV_6
\end{aligned} \tag{11}$$

where f^1, f^2 represent the feedback bytes. If the IV bytes in the right-hand side of (9), (10) and (11) are taken from a correct IV, then (6) will hold. In that case, also, the feedback bytes will be $f^1 = IV_0$ and $f^2 = IV_3 \oplus 0x33$. The first three steps of the F function when a correct IV is used are represented schematically in Figure 3.

Then, the filtering procedure for recovering k_j^i, $0 \leq i \leq 2$, $0 \leq j \leq 3$ amounts to substituting the F function key guesses into (8) along with the iv bytes derived from a correct IV and then verifying whether (6) holds. In particular, the filtering procedure is done round by round. As for the first F round, (6) amounts to $R^1 \gg 24 = 0x33$ and thus a candidate for $k^0 = k_3^0|k_2^0|k_1^0|k_0^0$ passes the criterion with probability 2^{-8}, which implies that 5 correct IVs are sufficient to uniquely determine k^0 with a good probability. We have verified experimentally that there is enough diffusion in one F-round to find the key uniquely with just 5 correct IVs.

As for the second step of the initialization phase, where (8) is executed for $i = 1$, first it should be noted that R^1 is known for each IV since $k_3^0|k_2^0|k_1^0|k_0^0$ is known. According to (6), the second F round criterion amounts to $R^2 \gg 28 \neq 3$. Thus, a guess for $k^1 = k_3^1|k_2^1|k_1^1|k_0^1$ passes the criterion with probability $\frac{15}{16}$. Assuming that all the wrong key bits can be eliminated, around 332 correct IV values will be required, since $2^{32} \times (\frac{15}{16})^{332} \approx 1$. In the previous section, 360 correct IVs has been generated, which ensures the unique recovery of k^1 with good probability. Throughout all our experiments, the number of candidates for k^1 that pass the test was consistently equal to 16. Without going into why 16 candidates always pass the test, it is noted that these candidates can be eliminated during the third F round filtering. The third F round criterion is $R^3 \gg 28 = 3$ and one can expect that the candidate for $k^2 = k_3^2|k_2^2|k_1^2|k_0^2$ passes with probability 2^{-4}, meaning that around 8 correct IV values will be required. The filtering is done for each of the 16 candidates for k^1. Again, experimentally, it was found that 16 candidates for k^2 always pass the test and therefore there will be 16 candidates at the end of the filtering procedure.

It remains to state how the correct IVs are drawn from L_i, $0 \leq i \leq 4$ to derive the iv^i values specified by (9), (10) and (11). For the first F round filtering, the 5 IVs are chosen from L_0, L_1, L_2, L_3 and L_4, respectively, which ensures that different 5 iv^0 values will be derived and that the filtering procedure will properly work. The second and third round choice of the IVs is arbitrary.

Attack Complexity: After the filtering procedure described above, there will remain 16 candidates for k_j^i, $0 \leq i \leq 2$, $0 \leq j \leq 3$ (96 bits). Each of the 16 candidates yields a linear system in the cipher key bytes determined by (9), (10) and (11). Since the linear equations in the system are independent, it follows that a $96 - 4 = 92$-bit constraint on the key K is specified. At this point, the attacker can either brute-force the remaining $128 - 92 = 36$ key bits or continue with the filtering process described above to deduce more key bits. In case of brute-forcing the 36 bits, the total complexity of the attack is dominated by around 2^{36} Loiss initialization procedures.

In the case where the filtering process is continued, the criterion $R^4 \gg 28 \neq 3$ can be used to filter out more key bits. Namely, expanding the corresponding iv^3 and k^3 values in a way analogous to (9)-(11), while taking into account the feedback byte in the LFSR update, reveals that altogether 20 more key bits can be recovered. In that case, the total complexity is dominated by the complexity of the above filtering procedures. The most expensive step is the filtering based on the second F round. We recall that in this filtering step, for each of the 360 correct IVs, each 32-bit key value is tested and eliminated if $R^2 \gg 28 \neq 3$ does not hold. Instead of applying the F function $2^{32} \times 360 \approx 2^{40.5}$ times, one can go through all key candidates for a particular IV, eliminate around $\frac{15}{16}$ of them and then, for the next IV, only go through the remaining candidates. In such a case, the number of applications of F is $\sum_{i=0}^{360}(\frac{15}{16})^i 2^{32} \approx 2^{36}$. To have further optimization, a table containing 2^{32} entries and representing F function can be prepared in advance. To measure the computational complexity of the attack in terms of Loiss initializations, a conservative estimate that one table lookup costs around 2^{-4} of a reasonable implementation of one Loiss initialization step could be accepted. Then, since there are $64 = 2^6$ steps in the initialization, the final complexity amounts to around 2^{26} Loiss initializations, $2^{25.8}$ chosen-IVs for both keys, storage space of 2^{32} 32-bit words and offline precomputation of 2^{32} applications of F, which is less than 2^{26} Loiss initializations, since each Loiss initialization includes 2^6 F computations.

Our attack was implemented and tested on a PC with 3 GHz Intel Pentium 4 processor with one core. Our implementation takes less than one hour to recover 92 bits of the key information and the attack procedure was successful on all the tested 32 randomly generated keys.

3.5 Towards a Resynchronization Attack

Here, some preliminary observations on the possibility of adapting the above attack to the single-key model are provided. In the single-key resynchronization attack, only the IV can have active bytes, which means that only the left-hand half of the LFSR, i.e., registers $s_{16}, \ldots, s_{31}$ as well as the BOMM will contain active bytes. As in the related-key attack above, the strategy is to have the difference cancelled out in the LFSR and localized only in the BOMM early during the initialization. One of the obstacles is that the R register will necessarily be activated when the difference reaches byte s_7, since the left-hand half of the LFSR contains active bytes. We note that this obstacle can be bypassed by cancelling the introduced R difference by having more than one LFSR byte active. Let LFSR bytes s_9, s_8 and s_7 be active with differences δ_2, δ_1, δ_0 at some time t during the initialization procedure. Also, assume that the word R and the BOMM bytes to be used in the next three steps are inactive. Below, we determine how many of the $(\delta_2, \delta_1, \delta_0)$ values can leave R inactive after 3 steps (after having passed through s_7) and also the probability of occurrence of such an event. For this purpose, note that the R cancellation event occurs if

$$\gamma(F(x^t) \oplus u^{t+1}) \oplus \gamma(F(x^t \oplus \delta_0) \oplus u^{t+1} \oplus \delta_1) = \theta^{-1}\delta_2 \tag{12}$$

where $x^t = R^t \oplus u^t$ and u^t denotes the 32-bit words fed to the F function from the LFSR in t-th step. By using a precomputed table for the S-box S_1 that, for each input and output difference, contains the information whether it is possible to achieve the input-output difference pair or not, we exhaustively checked for which values of $(\delta_2, \delta_1, \delta_0)$ equation (12) has solutions in x^t and u^{t+1}. The result of the finding is that only $2^{-12.861}$ of $(\delta_2, \delta_1, \delta_0)$ values cannot yield an R difference cancellation event. For the remaining $(\delta_2, \delta_1, \delta_0)$, for which (12) does have a solution, the probability of the R difference cancellation is $2^{-4} \times 2^{-28} = 2^{-32}$.

The analysis above indicates that attackers can choose almost any $(\delta_2, \delta_1, \delta_0)$ starting difference at three consecutive LFSR bytes and then bypass an R activation with a probability of 2^{-32}. A possible favorable position to introduce such $(\delta_2, \delta_1, \delta_0)$ difference can be in registers s_{18}, s_{17}, s_{16}, since the R register will only be activated through byte s_7. This can be done by activating IV_2, IV_1, IV_0 bytes. The 3-byte difference that arises in the BOMM then needs to be used for cancellations whenever some of the active LFSR bytes pass through the taps. Due to the relatively high number of cancellations that need to happen as the difference moves towards the right, we have not been able to bring the cancellation probability sufficiently high enough to have a practical attack. Controlling the difference propagation as done in [6] may be useful for that purpose. It is left for future research to verify whether a practical resynchronization single-key attack can be mounted against Loiss.

4 Sliding Properties of Loiss

In [5], a slide attack on SNOW 3G and SNOW 2.0 was provided. This attack is a related-key attack and involves a key-IV pair (K, IV) and (K', IV'). The idea is to have the inner state of the (K, IV) instance after $n \geq 1$ steps be a *starting* inner state. Then, the corresponding (K', IV') initializes to this starting state and the equality of the inner states is preserved until the end of the procedure. The similarity between the two keystreams is detected and this provides a basis for the key-recovery attack. Since LFSR-based word-oriented stream ciphers usually do not use counters which are the usual countermeasure against this kind of slide attacks, one way to protect against sliding is to have the initial inner state populated by the key, IV and constants so that it disallows the next several states to be starting states. For example, in ZUC [1], constants are loaded in a way that makes it difficult to mount a slide attack.

In the following, we point out that Loiss, similar to SNOW 2.0 and SNOW 3G, does not properly defend against sliding. If $C_0 = S_1^{-1}(0)$ and $C_1 = S_2(0)$, a slide by one step can be achieved as follows.

Observation 2. *Let $K = (K_{15}, \ldots, K_0)$ and $IV = (A, \ldots, A, B)$, where*

$$A = (\alpha \oplus \alpha^{-1} \oplus 1)^{-1}(K_0 \oplus \alpha^{-1} K_0 \oplus \alpha^{-1} K_1 \oplus K_2 \oplus \alpha K_5 \oplus \alpha K_8 \oplus K_{11} \oplus K_{13} \oplus C_0)$$

and B is determined by $B \oplus C_1 \oplus S_2(B \oplus C_1) = A$. Also, assume that $K_7 = C_0$ and $K_4 = K_{10} = K_{15} = C_0 \oplus A$. Then, for $K' = (K_0 \oplus B, K_{15}, \ldots, K_1)$ and $IV' = (A, \ldots, A)$, we have

$$z'_0 = z_1 \tag{13}$$

The proof of the observation is given in Appendix B.

Due to the requirement on bytes K_7, K_4, K_{10} and K_{15} from the formulation of the observation above, a Loiss key K has a related key pair specified by the observation above with probability 2^{-32}. For the related keys K and K' satisfying the conditions above, the attack can be performed by going through all $A \in F_2^8$ and verifying whether the relation (13) is satisfied for $IV = (A, \ldots, A, B)$, and $IV' = (A, \ldots, A)$. If yes, then such an A byte is a candidate for the right-hand side of the equation above specifying A, which depends only on K bytes. Each false candidate out of 2^8 candidates for A will pass the test (13) with probability 2^{-8}. That way, around one byte of the key information leaks. Slides by more than one step may also be possible.

5 Conclusion

We presented a practical-complexity related-key attack on the Loiss stream cipher. The fact that a slowly changing array (the BOMM) has been added as a part of the FSM in Loiss allowed the difference to be contained (i.e., do not propagate) during a large number of inner state update steps with a relatively high probability. The attack was implemented and our implementation takes less than one hour on a PC with 3GHz Intel Pentium 4 processor to recover 92 bits of the 128-bit key. The possibility of extending the attack to a resynchronization attack in a single-key model was discussed. We also showed that a slide attack is possible for the Loiss stream cipher.

References

1. Specification of the 3GPP Confidentiality and Integrity Algorithms 128-EEA3 and 128-EIA3. Document 2: ZUC Specification (2010), `http://www.dacas.cn`
2. Specification of SMS4, Block Cipher for WLAN Products - SMS4, Declassified (September 2006), (in Chinese) `http://www.oscca.gov.cn/UpFile/200621016423197990.pdf`
3. ETSI/SAGE. Specification of the 3GPP Confidentiality and Integrity Algorithms UEA2&UIA2 Document 2: SNOW 3G Specification (version 1.1) (September 2006), `http://www.3gpp.org/ftp`
4. Feng, D., Feng, X., Zhang, W., Fan, X., Wu, C.: Loiss: A Byte-Oriented Stream Cipher. In: Chee, Y.M., Guo, Z., Ling, S., Shao, F., Tang, Y., Wang, H., Xing, C. (eds.) IWCC 2011. LNCS, vol. 6639, pp. 109–125. Springer, Heidelberg (2011)

5. Kircanski, A., Youssef, A.: On the Sliding Property of SNOW 3G and SNOW 3.0. IET Information Security 4(5), 199–206 (2011)
6. Knellwolf, S., Meier, W., Naya-Plasencia, M.: Conditional Differential Cryptanalysis of Trivium and KATAN. In: Miri, A., Vaudenay, S. (eds.) SAC 2011. LNCS, vol. 7118, pp. 200–212. Springer, Heidelberg (2012)
7. Lin, D.,Jie, G.: Cryptanalysis of Loiss Stream Cipher. To appear in: The Computer Journal (2012), http://comjnl.oxfordjournals.org/content/early/2012/05/21/comjnl.bxs047.short?rss=1

A The Set of Possible Differences

Observation 1 is true for the following values (shown in hexadecimal):

$$\Delta = \{2, 5, 7, 9, d, 10, 11, 13, 15, 16, 18, 19, 1a, 1c, 1d, 1f, 20, 21, 25, 27, 2a, 2b, 2c, 2e, 2f, 31, 32, 37, 38, 39, 3d, 3e, 45, 48, 4a, 4b, 4d, 4f, 50, 54, 56, 57, 5b, 5c, 5d, 60, 61, 63, 64, 65, 66, 69, 6a, 6b, 6c, 6f, 70, 72, 74, 75, 77, 79, 7a, 7b, 7d, 7f, 80, 81, 82, 87, 89, 8b, 8d, 8e, 92, 94, 96, 97, 98, 99, 9a, 9c, 9d, 9e, a0, a1, a9, aa, ac, ae, af, b0, b2, b5, b8, ba, bc, bd, bf, c0, c1, c3, c4, c5, c7, ca, cd, d1, d2, d3, d4, d6, d7, d8, da, dc, de, df, e1, e2, e8, eb, ed, f0, f1, f2, f3, f4, f7, f9, fb, fc, ff\}$$

B Proof of Observations 1 and 2

In this appendix, we provide proofs for the two observations listed in the paper.
Proof of Observation 1:

From the cipher specification, $w^0 = 0x00$ is true regardless of the condition on the left-hand side. The two directions of the proof are provided as follows.
($\Leftarrow$): The change of the difference in the BOMM is described in Figure 2. In the first step, since $w^0 = 0x00$, both the value and the difference of y_3^0 remain unchanged and the LFSR difference is moved from s_3 to s_2. Since $w^1 = 0x33$ and both s_2 and y_3^1 are active with the same difference, they cancel out and the corresponding LFSR byte becomes inactive. As for the LFSR difference, it is just moved to s_1. Another effect of the second step is the change of the difference in y_3 byte from $0x02$ to $\alpha^{-1} \times 2$. Namely, expanding the difference in the y_3 byte and substituting the initial choice of $y_3^0 = 0x9d$ and also the choice of the starting difference $\delta = 0x02$ gives

$$y_3^2 \oplus y_3^{'2} = \delta \oplus S_2(y_3^0 \oplus S_2(0x33)) \oplus S_2(y_3^0 \oplus \delta \oplus S_2(0x33)) = \alpha^{-1} \times 0x02 \quad (14)$$

The third step moves the s_1 active byte to s_0, since $w^2 \gg 4 \neq 3$ and leaves the y_3 difference unchanged. Finally, since $w^3 \gg 4 = 0x3$, the difference in y_3 cancels out the difference in the LFSR update function (4) in the fourth step and this direction of the proof follows.

($\Rightarrow$): Clearly, $w^1 \gg 4 = 0x3$ since otherwise s_{31}^1 would be active and the LFSR after 4 steps would necessarily have at least one active byte. Moreover, $K = w^2 \gg 4 \neq 0x3$ holds since y_3^2 is necessarily active and otherwise there would be a difference introduced to the LFSR on byte s_{31}^2.

To show that $w^1 \bmod 4 = 0x3$, assume the contrary. In that case, the full LFSR cancellation in the fourth step cannot happen. Namely, in the second step, the difference in register y_3^1 remains unchanged, i.e., it remains equal to $0x02$. Therefore, during the third step, the existing one byte difference in the BOMM has to evolve to $\alpha^{-1} \times 2$ in order for the LFSR cancellation to happen in the fourth step. However, according to the S_2 specification, the input S_2 difference $0x02$ cannot be transformed to the output difference $\alpha^{-1} \times 2$ and thus $w^1 \bmod 4 = 0x3$.

Now, according to the ($\Leftarrow$) direction of the proof, (14) holds. To show that $w^3 \gg 4 = 0x3$, suppose the contrary. Since the LFSR byte s_0 is active at the fourth step (with the difference $0x2$), for this difference to be cancelled out, the BOMM output byte at step four has to be active with the same difference. Thus, the difference in y_3^2 which is equal to $\alpha^{-1} \times 0x02$ has to remain $\alpha^{-1} \times 0x02$ after passing through the S_2 S-box. This difference will necessarily be induced on some other BOMM byte since $K \neq 3$. However, such a possibility is ruled out by the S_2 specification: the S_2 S-box cannot map the input difference $\alpha^{-1} \times 2$ to $\alpha^{-1} \times 2$ output difference. It should be noted that this was possible in (14), since the same byte was updated twice in step 1. Therefore, $w^3 \gg 4 = 0x3$ has to hold. □

Proof of Observation 2

We will show that

$$\begin{aligned} IS^1 &= (s_{31}^1, \ldots, s_0^1, R^1, y_{16}^1, \ldots, y_0^1) \\ &= (s_{31}^{'0}, \ldots, s_0^{'0}, R^{'0}, y_{16}^{'0}, \ldots, y_0^{'0}) = IS^{'0} \end{aligned} \tag{15}$$

As for the BOMM bytes y_i, $15 \leq i \leq 0$, in the (K, IV) instance of the cipher, only y_0 will be updated since $R^0 = 0$. In other words, $y_i^1 = A$ for $15 \leq i \leq 1$. Moreover, from the specification of B, it follows that $y_0^1 = A$. Since $IV' = (A, \ldots, A)$, $y_i^{'0} = A$ for $15 \leq i \leq 0$ as well, i.e., (15) holds for the BOMM bytes. As for the equality between R^1 and $R^{'0}$, by the initialization procedure, $R^{'0} = 0$. To have $R^1 = 0$ as well, it suffices to have each of the four LFSR registers $s_{31}^0, s_{26}^0, s_{20}^0, s_7^0$ equal to $C_0 = S^{-1}(0)$, which is exactly the case due to the values to which bytes K_{15}, K_8, K_4 and K_7 are set. Finally, to establish the equality of the LFSR values in (15), the expression defining A are substituted into the way the LFSR is updated during the initialization procedure with the feed-forward, verifying that $s_{31}^1 = s_{31}^{'0} = K_{15} \oplus A$. As for the other LFSR values, $s_i^1 = s_i^{'0}$ holds directly due to the specification of K, IV, K', IV'.

Thus, the initialization procedures of the two cipher instances are slided, i.e., $IS^t = IS^{'t-1}$ for $1 \leq t \leq 64$. At time $t = 64$, in the (K, IV) instance of the cipher, a regular keystream step is applied, whereas in the (K', IV') instance, an initialization step is applied which destroys the slide property by introducing a difference between s_{31}^{65} and $s_{31}^{'64}$. However, it can be verified that this difference does not affect the two corresponding first keystream words, which proves (13). □

It should be noted that, as we verified by solving $B \oplus C_1 \oplus S_2(B \oplus C_1) = A$ for each $A \in F_2^8$, there always exists a byte B specified by this observation.

C Distinguisher Performance for Different (n, m)

The following table shows the numerical values for false positive and false negative probabilities for the distinguisher presented in section 3.2.

Table 1. Effectiveness of the distinguisher for different (n, m) parameters

(n, m)	P[false positive] $\approx$	P[false negative] $\approx$
$(16, 6)$	$2^{-35.1}$	$2^{-22.41}$
$(16, 8)$	$2^{-50.4}$	$2^{-16.00}$
$(24, 8)$	$2^{-44.6}$	$2^{-24.01}$
$(24, 10)$	$2^{-59.2}$	$2^{-19.91}$
$(32, 10)$	$2^{-54.2}$	$2^{-27.6}$
$(32, 12)$	$2^{-68.3}$	$2^{-20.68}$

Efficient Arithmetic on Elliptic Curves over Fields of Characteristic Three

Reza R. Farashahi[1,2,⋆], Hongfeng Wu[3,⋆⋆], and Chang-An Zhao[4,⋆⋆⋆]

[1] Dept. of Computing, Macquarie University, Sydney, NSW 2109, Australia
[2] Dept. of Mathematical Sciences, Isfahan University of Technology, 84156-83111 Isfahan, Iran
reza.farashahi@mq.edu.au, farashahi@cc.iut.ac.ir
[3] College of Sciences, North China University of Technology, Beijing 100144, China
whfmath@gmail.com
[4] School of Computer Science and Educational Software, Guangzhou University, Guangzhou 510006, China
changanzhao@gzhu.edu.cn

Abstract. This paper presents new explicit formulae for the point doubling, tripling and addition for ordinary Weierstraß elliptic curves with a point of order 3 and their equivalent Hessian curves over finite fields of characteristic three. The cost of basic point operations is lower than that of all previously proposed ones. The new doubling, mixed addition and tripling formulae in projective coordinates require $3\mathbf{M} + 2\mathbf{C}$, $8\mathbf{M} + 1\mathbf{C} + 1\mathbf{D}$ and $4\mathbf{M} + 4\mathbf{C} + 1\mathbf{D}$ respectively, where $\mathbf{M}$, $\mathbf{C}$ and $\mathbf{D}$ is the cost of a field multiplication, a cubing and a multiplication by a constant. Finally, we present several examples of ordinary elliptic curves in characteristic three for high security levels.

Keywords: Elliptic curve, Hessian curve, scalar multiplication, cryptography.

1 Introduction

Elliptic curve cryptosystem which was discovered by Neal Koblitz [13] and Victor Miller [16] independently requires smaller key sizes than the other public cryptosystems such as RSA at the same level of security. For example, a 160-bit elliptic curve key is competitive with a 1024-bit RSA key at the AES 80-bit security level. Thus it may be advantageous to use elliptic curve cryptosystems in resource-constrained environments, such as smart cards and embedded devices.

Scalar multiplication is a central operation in elliptic curve cryptographic schemes. There are numerous investigations of fast point multiplication on elliptic curves over large prime fields or binary fields. We refer to [3,8,1] for the

⋆ Supported in part by ARC Grant DP1092835.

⋆⋆ Corresponding author.'Supported in part by the National Natural Science Foundation of China (No. 11101002).

⋆⋆⋆ The work of C.-A Zhao is supported by the National Natural Science Foundation of China under Grants No. 61003227.

two cases. Note that ordinary elliptic curves in characteristic three could be applied in cryptographic schemes. For example, Koblitz implemented the digital signature algorithm on a special family of supersingular elliptic curves in characteristic three with great efficiency [14]. Compared to elliptic curves on large prime fields or binary fields, Smart *et al.* first pointed out that ordinary elliptic curve in characteristic three can be an alternative for implementing elliptic curve cryptosystems [20]. Recently, the improved formulae on this case are given in [17,12]. In [10], Hisil *et al.* gave a new tripling formulae for Hessian curve in characteristic three. The generalized form of Hessian curves has been presented by Farashahi, Joye, Bernstein, Lange and Kohel [6,4].

The goal of the present work is to speed up scalar multiplication on ordinary elliptic curves in characteristic three. We study the ordinary Weierstraß elliptic curves with a point of order 3 and their birationally equivalent Hessian curves over finite fields of characteristic 3.

The main contribution of this paper is given as follows:

- A modified projective coordinate system is presented for the Weierstraß elliptic curves with a rational point of order 3 over finite fields of characteristic 3. It is named as the scaled projective coordinate system which offers better performance than other projective coordinate systems.
- The basic point operations of addition, doubling, and tripling are investigated in the new scaled coordinate system for Weierstraß curves. The proposed formulae are faster than the previous known results.
- The new tripling formulae are presented for Hessian curves over finite fields of characteristic 3.
- The doubling and tripling formulae are complete for all input points in the rational group of these curves.

The paper is organized as follows. §2 recalls the necessary background for Weierstraß curves and Hessian curves over the finite fields $\mathbb{F}_{3^m}$. §3 presents new doubling, addition and tripling formulae for Weierstraß elliptic curves over $\mathbb{F}_{3^m}$ with a point of order three. §4 presents the new addition and tripling formulae for Hessian curves. §5 gives the efficiency consideration and timing results and §6 concludes the paper.

2 Preliminaries

2.1 Weierstraß Elliptic Curves over $\mathbb{F}_{3^m}$

Elliptic curves over any field can be divided into two classes of ordinary and supersingular elliptic curves. Every ordinary elliptic curve over the finite field $\mathbb{F}_{3^m}$ can be written in the Weierstraß form $y^2 = x^3 + ax^2 + b$, where $a, b \in \mathbb{F}_{3^m}$ and $ab \neq 0$. It is known, [20], that every ordinary elliptic curve over $\mathbb{F}_{3^m}$ with a point of order three can be written in the form

$$\mathrm{E}_b : y^2 = x^3 + x^2 + b$$

where $b \in \mathbb{F}_{3^m}$.

The sum of two (different) points (x_1, y_1), (x_2, y_2) on E_b is the point (x_3, y_3) given by ([20])

$$x_3 = \lambda^2 - x_1 - x_2 - 1, \quad \text{and} \quad y_3 = \lambda(x_1 - x_3) - y_1, \tag{1}$$

where $\lambda = (y_2 - y_1)/(x_2 - x_1)$.

The doubling of the point (x_1, y_1) on E_b is the point (x_3, y_3) given by ([20])

$$x_3 = \lambda^2 + x_1 - 1, \quad \text{and} \quad y_3 = \lambda(x_1 - x_3) - y_1, \tag{2}$$

where $\lambda = x_1/y_1$. Also, the inverse of the point (x_1, y_1) on E_b is the point $(x_1, -y_1)$. Furthermore, the tripling of the point (x_1, y_1) on E_b is the point (x_3, y_3) given by ([20])

$$x_3 = \frac{(x_1^3 + b)^3 - bx_1^3}{(x_1 + b)^2}, \quad \text{and} \quad y_3 = \frac{y_1^9 - y_1^3(x_1^3 + b)^2}{(x_1 + b)^3}. \tag{3}$$

Projective coordinate systems are preferred for point operations to avoid field inversions. There are some different types of projective coordinates which have the respective advantages in efficiency. The relationship between affine coordinates (x, y) and projective coordinates (X, Y, Z) is $(x, y) = (X/Z, Y/Z)$, for Jacobian projective coordinates, $(x, y) = (X/Z^2, Y/Z^3)$, and for López Dahab projective coordinates [15], $(x, y) = (X/Z, Y/Z^2)$.

2.2 Hessian Curves over $\mathbb{F}_{3^m}$

A *Hessian curve* over a finite field $\mathbb{F}_{3^m}$ is given by the cubic equation

$$\mathrm{H}_d: \ u^3 + v^3 + 1 = duv\,, \tag{4}$$

for some $d \in \mathbb{F}_{3^m}$ with $d \neq 0$ [9]. Furthermore, the generalized form of Hessian curves, called twisted Hessian as well, have been studied in [6,4]. A generalized Hessian curve $\mathrm{H}_{c,d}$ over $\mathbb{F}_{3^m}$ is defined by the equation

$$\mathrm{H}_{c,d} \ : \ u^3 + v^3 + c = duv \ ,$$

where $c, d \in \mathbb{F}_{3^m}$ with $c, d \neq 0$. Clearly, a Hessian curve H_d is a generalized Hessian curve $\mathrm{H}_{c,d}$ with $c = 1$. Furthermore, the generalized Hessian curve $\mathrm{H}_{c,d}$ over $\mathbb{F}_{3^m}$, via the map $(u, v) \mapsto (\widetilde{u}, \widetilde{v})$ given by $\widetilde{u} = u/\zeta$, $\widetilde{v} = v/\zeta$, with $\zeta = \sqrt[3]{c}$, is isomorphic to the Hessian curve $\mathrm{H}_{\frac{d}{\zeta}} \ : \ \widetilde{u}^3 + \widetilde{v}^3 + 1 = \frac{d}{\zeta}\widetilde{u}\widetilde{v}$. So, the families of Hessian curves and generalized Hessian over $\mathbb{F}_{3^m}$ are the same. For simplicity, from now on we consider the family of Hessian curves over $\mathbb{F}_{3^m}$. Furthermore, we recall from [6, Theorem 5] that the number of $\mathbb{F}_{3^m}$-isomorphism classes of the family of Hessian (or generalized Hessian) curves over $\mathbb{F}_{3^m}$ is $3^m - 1$.

The sum of two (different) points (u_1, v_1), (u_2, v_2) on H_d is the point (u_3, v_3) given by

$$u_3 = \frac{{v_1}^2 u_2 - {v_2}^2 u_1}{u_2 v_2 - u_1 v_1} \quad \text{and} \quad v_3 = \frac{{u_1}^2 v_2 - {u_2}^2 v_1}{u_2 v_2 - u_1 v_1} \ .$$

The doubling of the point (u_1, v_1) on H_d is the point (u_3, v_3) given by

$$u_3 = \frac{v_1(1 - {u_1}^3)}{{u_1}^3 - {v_1}^3} \quad \text{and} \quad v_3 = \frac{u_1({v_1}^3 - 1)}{{u_1}^3 - {v_1}^3} \ .$$

Also, the inverse of the point (u_1, v_1) on H_d is the point (v_1, u_1).

The projective closure of the curve H_d is

$$\mathbf{H}_d : \ U^3 + V^3 + W^3 = dUVW \ .$$

The neutral element of the group of $\mathbb{F}$-rational points of $\mathbf{H}_d$ is the point at infinity $(1, -1, 0)$ and the inverse of the point $P = (U_1, V_1, W_1)$ on $\mathbf{H}_d$, is the point $-P = (V_1, U_1, W_1)$.

The sum of the points (U_1, V_1, W_1), (U_2, V_2, W_2) on $\mathbf{H}_d$ is the point (U_3, V_3, W_3) with

$$U_3 = U_2W_2{V_1}^2 - U_1W_1{V_2}^2, \quad V_3 = V_2W_2{U_1}^2 - V_1W_1{U_2}^2, \quad W_3 = U_2V_2{W_1}^2 - U_1V_1{W_2}^2 \ . \tag{5}$$

The doubling of the point (U_1, V_1, W_1) on $\mathbf{H}_d$ is the point (U_3, V_3, W_3) given by

$$U_3 = V_1({W_1}^3 - {U_1}^3), \quad V_3 = U_1({V_1}^3 - {W_1}^3), \quad W_3 = W_1({U_1}^3 - {V_1}^3) \ . \tag{6}$$

We note that the addition formulae (5) is not *unified*, i.e., the formulae do not work to double a point. The following set of formulae are unified which make Hessian curves interesting against side-channel attacks [1,2].

The sum of the points (U_1, V_1, W_1) and (U_2, V_2, W_2) on $\mathbf{H}_d$ is the point (U_3, V_3, W_3) given by

$$U_3 = V_2W_2{W_1}^2 - U_1V_1{U_2}^2, \quad V_3 = U_2V_2{V_1}^2 - U_1W_1{W_2}^2, \quad W_3 = U_2W_2{U_1}^2 - V_1W_1{V_2}^2 \ . \tag{7}$$

Furthermore, by swapping the order of the points in the addition formulae (7), we obtain the following unified formulae.

$$U_3 = V_1W_1{W_2}^2 - U_2V_2{U_1}^2, \quad V_3 = U_1V_1{V_2}^2 - U_2W_2{W_1}^2, \quad W_3 = U_1W_1{U_2}^2 - V_2W_2{V_1}^2 \ . \tag{8}$$

We recall [6, Propositions 1], which describes the exceptional cases of the addition formulae (5).

Proposition 1. *The addition formulae* (5) *work for all pairs of points P_1, P_2 on $\mathbf{H}_d$ if and only if $P_1 - P_2$ is not the point at infinity.*

Since the curve $\mathbf{H}_d$ over $\mathbb{F}_{3^m}$ has only one $\mathbb{F}_{3^m}$-rational point at infinity, the addition formulae (5) work for all *distinct* pairs of $\mathbb{F}_{3^m}$-rational inputs.

We recall [6, Propositions 2], that explains the exceptional cases of the addition formulae (7).

Proposition 2. *The addition formulae* (7) *work for all pairs of points* P_1, P_2 *on* $\mathbf{H}_d$ *if and only if* $P_1 - P_2 \neq (-1, 0, 1)$.

Similarly, the addition formulae (8) work for all pairs of points P_1, P_2 on $\mathbf{H}_d$ if and only if $P_1 - P_2$ is not a 3-torsion point of $\mathbf{H}_d$ with X-coordinate equals 0. So, the set of formulae (7) and (8) are complement of each other, i.e., if formulae (7) do not work for the pair of inputs P_1, P_2, then the other do work.

As a consequence, the doubling formulae (6) work for all points of the curve $\mathbf{H}_d$. Moreover, for the subgroup $\mathcal{H}$ of $\mathbf{H}_d(\mathbb{F}_{3^m})$ not including the point $(-1, 0, 1)$, the addition formulae (7) (and (8)) work for all pairs of points in $\mathcal{H}$.

2.3 Birational Equivalence

We note that every Hessian curve H_d over $\mathbb{F}_{3^m}$ has a point of order 3. Moreover, every elliptic curve over $\mathbb{F}_{3^m}$ with a point of order 3 can be given in generalized Hessian form (see [6]) and so in Hessian form. From §2.1, we recall that an ordinary elliptic curve over $\mathbb{F}_{3^m}$ has a point of order 3 if and only if it can be given by the equation $y^2 = x^3 + x^2 + b$, for some $b \in \mathbb{F}_q$. Therefore, we have the birational equivalence between these two forms.

The ordinary elliptic curve E_b in Weierstraß form $\mathrm{E}_b : y^2 = x^3 + x^2 + b$, where $b \neq 0$, via the map $(x, y) \mapsto (u, v)$ defined by

$$x = d(u + v) \quad \text{and} \quad y = d(u - v)$$

is birationally equivalent to Hessian curve $\mathrm{H}_d : u^3 + v^3 + 1 = duv$, where $d^3 = -1/b$. The inverse map $(u, v) \mapsto (x, y)$ is given by

$$x = -(u + v)/d \quad \text{and} \quad y = -(u - v)/d.$$

In the projective model, the point (U, V, W) on the projective curve

$$\mathbf{H}_d : U^3 + V^3 + W^3 = dUVW,$$

is mapped to the point $(-(U + V), -(U - V), dW)$ on the projective Weierstraß curve

$$\mathbf{E}_b : ZY^2 = X^3 + X^2 Z + bZ^3,$$

where $b = -1/d^3$. Furthermore, via the inverse map, the point (X, Y, Z) on $\mathbf{E}_b$ is corresponded to the point $(X + Y, X - Y, Z/d)$ on $\mathbf{H}_d$. So, we suggest to use the *scaled* projective coordinate system (X, Y, T), where $dT = Z$ and (X, Y, Z) is a point on $\mathbf{E}_b$. Then, the scaled point (X, Y, T) on $\mathbf{E}_b$ is corresponded to the point $(X + Y, X - Y, T)$ on $\mathbf{H}_d$. Furthermore, via the inverse map, the point (U, V, W) on $\mathbf{H}_d$ is corresponded to the scaled point $(-(U + V), -(U - V), W)$ on $\mathbf{E}_b$.

3 Explicit Formulae for Ordinary Weierstraß Form

In this section, we show how to use a new projective coordinate system to speed up basic point operations on ordinary Weierstraß elliptic curves with a point of order 3 over finite fields of characteristic three.

Here, we consider elliptic curves in Weierstraß form

$$\mathbf{E}_b : Y^2Z = X^3 + X^2Z + bZ^3,$$

where $b \in \mathbb{F}_{3^m}, b \neq 0$. We let $b = -1/d^3$ for some $d \in \mathbb{F}_q$, i.e., $d = (\frac{-1}{b})^{3^{(m-1)}}$. We use the *scaled* projective system, where the point (X, Y, T) is a *scaled* point, if $T = Z/d$ and (X, Y, Z) is a projective point on $\mathbf{E}_{-1/d^3}$. We note that points $(1/d, \pm 1/d, 1)$ are the points of order three on $\mathbf{E}_{-1/d^3}$. The correspondence between the scaled projective coordinates and the affine coordinates is given as follows

$$(\frac{X}{dT}, \frac{Y}{dT}) \leftrightarrow (X, Y, T).$$

3.1 Point Doubling

Here, using the scaled projective coordinates system, we provide a new formulae for point doubling for the elliptic curve $\mathbf{E}_{-1/d^3} : \ Y^2Z = X^3 + X^2Z - Z^3/d^3$, where $d \in \mathbb{F}_{3^m}$.

Let (X_1, Y_1, T_1) be a scaled point on $\mathbf{E}_{-1/d^3}$, i.e., $T_1 = Z_1/d$ and (X_1, Y_1, Z_1) is a point on $\mathbf{E}_{-1/d^3}$. So, $dY_1^2T_1 = X_1^3 + dX_1^2T_1 - T_1^3$. Let $(X_3, Y_3, T_3) = [2](X_1, Y_1, T_1)$, which is the doubling in the scaled projective coordinates system. From the affine doubling formula (2), we have

$$X_3 = d(X_1^2Y_1 - Y_1^3)T_1 + X_1Y_1^3, \ Y_3 = d(X_1Y_1^2 - X_1^3)T_1 - Y_1^4, \ T_3 = T_1Y_1^3.$$

Then

$$\begin{aligned} X_3 &= X_1Y_1^3 - X_1^3Y_1 + Y_1(X_1^3 + dX_1^2T_1 - dY_1^2T_1), \\ Y_3 &= X_1(dY_1^2T_1 - dX_1^2T_1) - Y_1^4 = X_1(X_1^3 - T_1^3) - Y_1^4. \end{aligned}$$

Therefore, we obtain

$$X_3 = X_1Y_1^3 + Y_1T_1^3 - X_1^3Y_1, \ Y_3 = X_1^4 - Y_1^4 - X_1T_1^3, \ T_3 = T_1Y_1^3. \qquad (9)$$

The following algorithm computes (X_3, Y_3, T_3), i.e., the doubling of the point (X_1, Y_1, T_1).

$$\begin{aligned} &A = X_1 + Y_1, \ B = X_1 - Y_1, \ D = (T_1 - A)^3, \\ &E = (B - T_1)^3, \ F = B \cdot D, \ G = A \cdot E, \ H = T_1 \cdot (D + E), \\ &X_3 = F + G, \ Y_3 = F - G, \ T_3 = H. \end{aligned}$$

The cost of above algorithm is $3\mathbf{M} + 2\mathbf{C}$, where $\mathbf{M}$ is the cost of a field multiplication and $\mathbf{C}$ is the cost of cubing.

The following proposition shows that the doubling formulae is complete.

Proposition 3. *The doubling formulae (9) work for all input points on* $\mathbf{E}_{-1/d^3}$.

Proof. Let $P = (X_1, Y_1, T_1)$ be a scaled point on $\mathbf{E}_{-1/d^3}$ such that the doubling formulae (9) do not work for the input P. Thus, we have

$$X_3 = X_1Y_1^3 + Y_1T_1^3 - X_1^3Y_1 = 0, \ Y_3 = X_1^4 - Y_1^4 - X_1T_1^3 = 0, \ T_3 = T_1Y_1^3 = 0.$$

From $T_3 = 0$, we have $T_1 = 0$ or $Y_1 = 0$. By the curve equation we have $dY_1^2T_1 = X_1^3 + dX_1^2T_1 - T_1^3$. If $T_1 = 0$ then $X_1 = 0$. From $Y_3 = 0$, we obtain $Y_1 = 0$. So $(X_1, Y_1, T_1) = (0, 0, 0)$ which is a contradiction. If $Y_1 = 0$, by $Y_3 = 0$ we have $X_1(X_1 - T_1)^3 = 0$. Then, by the curve equation we obtain $T_1 = 0$, which is a contradiction. □

3.2 Point Addition

Now, we provide the addition formulae for the scaled points on $\mathbf{E}_{-1/d^3} : Y^2Z = X^3 + X^2Z - Z^3/d^3$

Let $P_1 = (X_1, Y_1, T_1)$ and $P_2 = (X_2, Y_2, T_2)$ be two scaled points on $\mathbf{E}_{-1/d^3}$, i.e., $T_1 = Z_1/d$, $T_2 = Z_2/d$ and (X_1, Y_1, Z_1), (X_2, Y_2, Z_2) are points on $\mathbf{E}_{-1/d^3}$. Let (X_3, Y_3, T_3) be the sum of P_1 and P_2, where $T_3 = Z_3/d$ and (X_3, Y_3, Z_3) is a point on $\mathbf{E}_{-1/d^3}$. From the affine addition formulae (1), we have

$$\begin{aligned}
X_3 &= dT_1T_2(X_2T_1 - X_1T_2)((Y_2T_1 - Y_1T_2)^2 - (X_2T_1 - X_1T_2)^2) \\
&\quad -(X_2T_1 - X_1T_2)^3(X_2T_1 + X_1T_2), \\
Y_3 &= -dT_1T_2(Y_2T_1 - Y_1T_2)((Y_2T_1 - Y_1T_2)^2 - (X_2T_1 - X_1T_2)^2) \\
&\quad +(X_2T_1 - X_1T_2)^3(Y_2T_1 + Y_1T_2), \\
T_3 &= T_1T_2(X_2T_1 - X_1T_2)^3.
\end{aligned}$$

Then, we obtain

$$\begin{aligned}
X_3 &= T_2(X_1^2X_2 + X_1Y_1Y_2 + X_2Y_1^2) - T_1(X_1X_2^2 + Y_1X_2Y_2 + X_1Y_2^2), \\
Y_3 &= T_2(X_1^2Y_2 + X_1Y_1X_2 + Y_2Y_1^2) - T_1(Y_1X_2^2 + X_1X_2Y_2 + Y_1Y_2^2), \\
T_3 &= T_1^2(X_2 + Y_2)(X_2 - Y_2) - T_2^2(X_1 + Y_1)(X_1 - Y_1).
\end{aligned} \tag{10}$$

The following addition algorithm performs the addition formulae (10), which requires 12**M**.

$$\begin{aligned}
&A_1 = X_1 + Y_1,\ B_1 = X_1 - Y_1,\ A_2 = X_2 + Y_2,\ B_2 = X_2 - Y_2, \\
&D = T_1 \cdot A_2,\ E = T_1 \cdot B_2,\ F = T_2 \cdot A,\ G = T_2 \cdot B, \\
&H = A_1 \cdot B_2,\ I = A_2 \cdot B_1,\ X_3 = G \cdot I - E \cdot H, \\
&Y_3 = F \cdot H - D \cdot I,\ T_3 = D \cdot E - F \cdot G.
\end{aligned}$$

Notice that $(T_1 - X_1)^3 = aT_1(X_1 + Y_1)(X_1 - Y_1)$. Then, we have

$$\begin{aligned}
&T_1T_2 \cdot (T_1^2(X_2 + Y_2)(X_2 - Y_2) - T_2^2(X_1 + Y_1)(X_1 - Y_1)) \\
&= (1/d)(T_1^3(T_2 - X_2)^3 - T_2^3(T_1 - X_1)^3) = (1/d)(X_1T_2 - X_2T_1)^3.
\end{aligned}$$

Therefore, we obtain

$$\begin{aligned}
X_3 &= T_2T_1^2(X_1X_2^2 + Y_1X_2Y_2 + X_1Y_2^2) - T_1T_2^2(X_1^2X_2 + X_1Y_1Y_2 + X_2Y_1^2), \\
Y_3 &= T_2T_1^2(Y_1X_2^2 + X_1X_2Y_2 + Y_1Y_2^2) - T_1T_2^2(X_1^2Y_2 + X_1Y_1X_2 + Y_2Y_1^2), \\
T_3 &= (1/d)(X_2T_1 - X_1T_2)^3.
\end{aligned} \tag{11}$$

We write

$$\begin{aligned}
X_3 &= T_1(X_2 + Y_2)T_2^2(X_1 - Y_1)^2 + T_1(X_2 - Y_2)T_2^2(X_1 + Y_1)^2 \\
&\quad -T_2(X_1 + Y_1)T_1^2(X_2 - Y_2)^2 - T_2(X_1 - Y_1)T_1^2(X_2 + Y_2)^2
\end{aligned}$$

and

$$\begin{aligned}Y_3 = T_1(X_2+Y_2)T_2^2(X_1-Y_1)^2 - T_1(X_2-Y_2)T_2^2(X_1+Y_1)^2\\ -T_2(X_1+Y_1)T_1^2(X_2-Y_2)^2 + T_2(X_1-Y_1)T_1^2(X_2+Y_2)^2.\end{aligned}$$

Therefore, we have the following addition algorithm which requires $10\mathbf{M}+1\mathbf{C}+1\mathbf{D}$, where $\mathbf{D}$ is the cost of a field multiplication by the constant $1/d$.

$$\begin{aligned}&A_1 = X_1+Y_1,\ B_1 = X_1-Y_1,\ A_2 = X_2+Y_2,\ B_2 = X_2-Y_2,\\ &D = B_1\cdot T_2,\ E = A_2\cdot T_1,\ F = A_1\cdot T_2,\ G = B_2\cdot T_1,\ H = D\cdot E\\ &I = F\cdot G,\ J = F\cdot I,\ K = E\cdot H,\ X_3 = D\cdot H + J - G\cdot I - K,\\ &Y_3 = X_3 + J + K,\ Z_3 = (1/d)(D+F-E-G)^3.\end{aligned}$$

The cost of mixed scaled addition formulae is $8\mathbf{M}+1\mathbf{C}+1\mathbf{D}$, by setting $T_1 = 1$.

3.3 Unified Addition Formulae

Here, we study the *unified* addition formulae. In general, the *unified* addition formulae work for all but finitely many pairs of points. The *complete* addition formulae emphasize to work for all inputs. We recall that the affine addition formulae (1) and projective formulae (11) do not work to double a point. More precisely, the addition formula (11) do not work for the points P_1 and P_2 if and only if $P_1 - P_2 = (0,1,0)$.

Hereafter, we give some *unified* addition formulae for $\mathbf{E}_{-1/d^3}: Y^2Z = X^3 + X^2Z - Z^3/d^3$. The unified addition formulae make the curve $\mathbf{E}_{-1/d^3}$ interesting against side-channel attacks.

Let $P_1 = (X_1,Y_1,T_1)$ and $P_2 = (X_2,Y_2,T_2)$ be two scaled points on $\mathbf{E}_{-1/d^3}$, where $T_1 = Z_1/d$, $T_2 = Z_2/d$ and (X_1,Y_1,Z_1), (X_2,Y_2,Z_2) are points on $\mathbf{E}_{-1/d^3}$. Then, $Q_1 = (X_1+Y_1, X_1-Y_1, T_1)$ and $Q_2 = (X_2+Y_2, X_2-Y_2, T_2)$ are points of H_d. From the unified formulae (7) we obtain the point (U_3,V_3,W_3) on H_d, where

$$\begin{aligned}U_3 &= T_1^2T_2(X_2-Y_2) - (X_1+Y_1)(X_1-Y_1)(X_2+Y_2)^2,\\ V_3 &= -T_1T_2^2(X_1+Y_1) + (X_2+Y_2)(X_2-Y_2)(X_1-Y_1)^2,\\ W_3 &= T_2(X_1+Y_1)^2(X_2+Y_2) - T_1(X_1-Y_1)(X_2-Y_2)^2.\end{aligned}$$

Then, the point $(X_3,Y_3,T_3) = ((U_3+V_3),(U_3-V_3),-W_3)$ is a scaled point of $\mathbf{E}_{-1/d^3}$, which is the sum of P_1 and P_2. We obtain

$$\begin{aligned}X_3 &= T_1T_2(T_1(X_2-Y_2) - T_2(X_1+Y_1)) + (X_1-Y_1)(X_2+Y_2)(X_1Y_2+X_2Y_1),\\ Y_3 &= T_1T_2(T_1(X_2-Y_2) + T_2(X_1+Y_1)) + (X_1-Y_1)(X_2+Y_2)(X_1X_2+Y_1Y_2),\\ T_3 &= T_1(X_1-Y_1)(X_2-Y_2)^2 - T_2(X_1+Y_1)^2(X_2+Y_2).\end{aligned} \tag{12}$$

We note that by swapping the order of the points P_1 and P_2 we obtain another unified formulae as follows.

$$\begin{aligned}X_3 &= T_1T_2(T_2(X_1-Y_1) - T_1(X_2+Y_2)) + (X_1+Y_1)(X_2-Y_2)(X_1Y_2+X_2Y_1),\\ Y_3 &= T_1T_2(T_2(X_1-Y_1) + T_1(X_2+Y_2)) + (X_1+Y_1)(X_2-Y_2)(X_1X_2+Y_1Y_2),\\ T_3 &= T_2(X_1-Y_1)^2(X_2-Y_2) - T_1(X_1+Y_1)(X_2+Y_2)^2.\end{aligned} \tag{13}$$

Moreover, the algorithm which performs above addition formulae (12) (or (13)) requires 12**M**.

We recall that the set of formulae (7) and (8) are complement of each other, so the same property is true for the set of formulae (12) and (13). From Proposition 2, we see that the addition formulae (12) do not work for the inputs P_1, P_2 if and only if $P_1 - P_2 = (1, 1, 1)$.

Then, one can easily see that the doubling formulae (6) work for all points of the curve $\mathbf{H}_d$. Moreover, for the subgroup $\mathcal{G}$ of $\mathbf{E}_{-1/d^3}(\mathbb{F}_{3^m})$ not including the point $(1, 1, d)$ (or the scaled point $(1, 1, 1)$), the addition formulae (12) (and (13)) work for all pairs of points in $\mathcal{G}$.

3.4 Point Tripling

When implementing scalar multiplication on elliptic curves over finite fields of characteristic three, it is convenient to choose a base three expansion for an exponent k since the cubing operation in the finite field is cheaper than other basic operations. Now point tripling is considered as follows.

From the affine tripling formulae (3), the tripling of the scaled point (X_1, Y_1, T_1) on $\mathbf{E}_{-1/d^3}$ is the point (X_3, Y_3, T_3) given by

$$\begin{aligned} X_3 &= (X_1^3 - T_1^3)(X_1^9 - T_1^9 + d^3 X_1^3 T_1^6), \\ Y_3 &= d^3 Y_1^3 T_1^3 (Y_1^2 - X_1^2 - T_1^2 - X_1 T_1)^3, \\ T_3 &= d^2 (X_1^9 T_1^3 - T_1^{12}). \end{aligned} \tag{14}$$

Then, we have

$$\begin{aligned} X_3 &= (X_1 - T_1)^3 (d^3 Y_1^6 T_1^3 - d^3 X_1^6 T_1^3 + d^3 X_1^3 T_1^6) \\ &= -d^3 T_1^3 \cdot (X_1 - T_1)^3 (X_1^6 - Y_1^6 - X_1^3 T_1^3) \\ &= -d^3 T_1^3 \cdot (X_1 - T_1)^3 (X_1^2 - Y_1^2 - X_1 T_1)^3, \\ Y_3 &= -d^3 T_1^3 \cdot Y_1^3 (X_1^2 + X_1 T_1 + T_1^2 - Y_1^2)^3, \\ T_3 &= -d^3 T_1^3 \cdot (T_1^9 - X_1^9)/d. \end{aligned}$$

So, we obtain

$$\begin{aligned} X_3 &= (X_1 - T_1)^3 (X_1^2 - Y_1^2 - X_1 T_1)^3, \\ Y_3 &= Y_1^3 (X_1^2 + X_1 T_1 + T_1^2 - Y_1^2)^3, \\ T_3 &= (T_1^9 - X_1^9)/d. \end{aligned} \tag{15}$$

We also write

$$X_1^2 + X_1 T_1 + T_1^2 - Y_1^2 = (X_1 - T_1 + Y_1)(X_1 - T_1 - Y_1),$$
$$X_1^2 - Y_1^2 - X_1 T_1 = (X_1^2 + X_1 T_1 + T_1^2 - Y_1^2) + X_1 T_1 - T_1^2 \ .$$

Then, we propose the following very fast point tripling algorithm.

$$A = X_1 - T_1,\ B = (A + Y_1)(A - Y_1),\ D = A(B + T_1 A),$$
$$X_3 = D^3,\ Y_3 = (Y_1 B)^3,\ T_3 = -(1/d) A^9 \ .$$

We see that the cost for above point tripling algorithm is $4\mathbf{M} + 4\mathbf{C} + 1\mathbf{D}$. The following proposition shows that tripling formulae work for all inputs.

Proposition 4. *The tripling formulae* (15) *work for all points on* $\mathbf{E}_{-1/d^3}$.

Proof. Let $P = (X_1, Y_1, T_1)$ be a scaled point on $\mathbf{E}_{-1/d^3} : Y^2Z = X^3 + X^2Z - Z^3/d^3$ such that the tripling formulae (15) do not work for the point P. Thus, the formulae (15) output

$$X_3 = 0,\ Y_3 = 0,\ T_3 = (T_1^9 - X_1^9)/d = 0.$$

From $T_3 = 0$, we have $X_1 = T_1$. Then, $Y_3 = Y_1^3(X_1^2 + X_1T_1 + T_1^2 - Y_1^2)^3 = -Y_1^5$. Since $Y_3 = 0$, we have $Y_1 = 0$ and then $(X_1, Y_1, T_1) = (0, 0, 0)$ which is a contradiction. □

4 Explicit Formulae for Hessian Curves in Characteristic 3

In this section, we present fast point addition and tripling formulae for Hessian curves over a field $\mathbb{F}$ of characteristic 3.

4.1 Addition and Doubling Formulae

The point addition algorithms for formulae (5) are described in [5,11,19] with the cost of 12**M**. Also, these addition formulae can be performed in a parallel way, see [19]. In particular, the addition formulae (5) in a parallel environment using 3, 4 or 6 processors require 4**M**, 3**M** or 2**M**, respectively.

Furthermore, from the addition formulae (5), the sum of the points $(X_1 : Y_1, Z_1)$, (X_2, Y_2, Z_2) on H_d is the point (X_3, Y_3, Z_3) given by

$$X_3 = Z_1Z_2(X_2Z_2{Y_1}^2 - X_1Z_1{Y_2}^2), \quad Y_3 = Z_1Z_2(Y_2Z_2{X_1}^2 - Y_1Z_1{X_2}^2),$$

$$\begin{aligned} Z_3 &= Z_1Z_2(X_2Y_2{Z_1}^2 - X_1Y_1{Z_2}^2) = Z_1^3(X_2Y_2Z_2) - Z_2^3(X_1Y_1Z_1) \\ &= Z_1^3(X_2^3 + Y_2^3 + Z_2^3)/d - Z_2^3(X_1^3 + Y_1^3 + Z_1^3)/d \\ &= (X_2Z_1 + Y_2Z_1 - X_1Z_2 - Y_1Z_2)^3/d. \end{aligned}$$

Using the next algorithm, the cost of above formulae is $10\mathbf{M} + 1\mathbf{C} + 1\mathbf{D}$, where $1\mathbf{D}$ is the cost of the multiplication by the constant $1/d$.

$$\begin{gathered} A = X_2Z_1,\ B = Y_2Z_1,\ C = X_1Z_2,\ D = Y_1Z_2,\ E = AD,\ F = BC, \\ X_3 = DE - BF, \quad Y_3 = CF - AE, \quad Z_3 = (1/d)(A + B - C - D)^3\ . \end{gathered} \tag{16}$$

And, the mixed addition formulae requires $8\mathbf{M}+1\mathbf{C}+1\mathbf{D}$. We also noticed that, Kim *et al.*, [12], proposed a mixed addition algorithm which requires $8\mathbf{M}+1\mathbf{C}+1\mathbf{D}$. But, none of above addition algorithms is unified.

The next algorithm evaluates the unified addition formulae (7) for the Hessian curve H_d with 12**M**.

$$\begin{gathered} A = X_1X_2,\ B = Y_1Y_2,\ C = Z_1Z_2,\ D = X_1Z_2,\ E = Y_1X_2,\ F = Z_1Y_2, \\ X_3 = CF - AE, \quad Y_3 = BE - CD, \quad Z_3 = AD - BF\ . \end{gathered}$$

The mixed addition formulae requires 10**M** by setting $Z_2 = 1$. Furthermore, the addition formulae (7) can be performed in a parallel way. The following addition algorithm is similar to the addition algorithm (16) which requires 10**M**+1**C**+1**D**.

$$A = Z_2X_1,\ B = X_2X_1,\ C = Y_1Y_2,\ D = Z_1Y_2,\ E = AD,\ F = BC,$$
$$X_3 = DE - BF, \quad Y_3 = CF - AE, \quad Z_3 = (1/d)(A + B - C - D)^3 \ .$$

Moreover, this addition algorithm is unified. Also, the cost of the mixed addition formulae is $8\mathbf{M} + 1\mathbf{C} + 1\mathbf{D}$ by setting $X_1 = 1$.

From the doubling formulae (6), the doubling of the point (X_1, Y_1, Z_1) on H_d is the point (X_3, Y_3, Z_3) given by

$$X_3 = Y_1(Z_1 - X_1)^3, \quad Y_3 = X_1(Y_1 - Z_1)^3, \quad Z_3 = Z_1(X_1 - Y_1)^3 \ .$$

which requires $3\mathbf{M} + 2\mathbf{C}$([12]).

4.2 Point Tripling

From §2.3, we recall that the scaled point (X, Y, T) on $\mathbf{E}_b$ is corresponded to the point $(X + Y, X - Y, T)$ on the Hessian curve $\mathbf{H}_d$. Furthermore, the point (U, V, W) on $\mathbf{H}_d$ is corresponded to the scaled point $((U + V), (U - V), -T)$ on $\mathbf{E}_b$. The point tripling (15) for Weierstraß form $\mathbf{E}_b$ can be used to obtain the following point tripling algorithm for the Hessian curve $\mathbf{H}_d$. The tripling of the points (U_1, V_1, W_1) on $\mathbf{H}_d$ with $d^3 = -1/b$ is the scaled point (U_3, V_3, W_3) given by the next formulae.

$$\begin{aligned} U_3 &= (U_1W_1^2 + V_1U_1^2 + W_1V_1^2)^3, \\ V_3 &= (U_1V_1^2 + V_1W_1^2 + W_1U_1^2)^3, \\ W_3 &= -(1/d)(U_1 + V_1 + W_1)^9. \end{aligned} \tag{17}$$

Then, we propose the following point tripling algorithm.

$$A = U_1 + V_1 + W_1,\ B = (U_1 - W_1)(V_1 - W_1),\ D = A(B - AZ_1),\ E = V_1B$$
$$U_3 = (D + E)^3,\ V_3 = (D - E)^3,\ W_3 = -(1/d)A^9 \ .$$

The cost for above point tripling algorithm is $4\mathbf{M} + 4\mathbf{C} + 1\mathbf{D}$. Moreover, from Proposition 4 we see that the tripling formulae work for all inputs.

5 Operation Count Comparison

The efficiency of implementing elliptic curve cryptosystems depends on the speed of basic point operations. In this section, we will compare the new formulae for point operations with the previously known results on the corresponding curve.

We first recall the previous results on ordinary curves in characteristic three. In [12], Kim *et al.* proposed a type of projective coordinate system (ML-coordinates) which consists of four variables and the relationship between this

Table 1. Costs of point operations for different coordinate systems of elliptic curves over $\mathbb{F}_{3^m}$

Coordinate System	Mixed addition	Doubling	Tripling
Projective[20]	$9\mathbf{M}+2\mathbf{S}+1\mathbf{C}$	$6\mathbf{M}+0\mathbf{S}+3\mathbf{C}$	$7\mathbf{M}+2\mathbf{S}+5\mathbf{C}$
Jacobian[20]	$7\mathbf{M}+3\mathbf{S}+2\mathbf{C}$	$6\mathbf{M}+2\mathbf{S}+3\mathbf{C}$	$5\mathbf{M}+1\mathbf{S}+4\mathbf{C}+1\mathbf{D}$
López Dahab[20]	$10\mathbf{M}+3\mathbf{S}$	$7\mathbf{M}+4\mathbf{S}+2\mathbf{C}$	$10\mathbf{M}+3\mathbf{S}+5\mathbf{C}$
Projective in Hessian form[20]	$10\mathbf{M}$	$3\mathbf{M}+3\mathbf{C}$	$-$
Projective in Hessian form[10]	-	-	$6\mathbf{M}+4\mathbf{C}+2\mathbf{D}$
Jacobian[17]	$7\mathbf{M}+3\mathbf{S}+2\mathbf{C}+1\mathbf{D}$	$5\mathbf{M}+2\mathbf{S}+3\mathbf{C}$	$3\mathbf{M}+2\mathbf{S}+5\mathbf{C}+1\mathbf{D}$
ML-coordinates [12]	$8\mathbf{M}+2\mathbf{C}$	$5\mathbf{M}+3\mathbf{S}+3\mathbf{C}$	$6\mathbf{M}+6\mathbf{C}$
Hessian form [12]	$9\mathbf{M}+1\mathbf{C}$	$3\mathbf{M}+2\mathbf{C}$	$-$
Hessian form(this work)	$8\mathbf{M}+1\mathbf{C}+1\mathbf{D}$	$3\mathbf{M}+2\mathbf{C}$	$4\mathbf{M}+4\mathbf{C}+1\mathbf{D}$
scaled projective(this work)	$8\mathbf{M}+1\mathbf{C}+1\mathbf{D}$	$3\mathbf{M}+2\mathbf{C}$	$4\mathbf{M}+4\mathbf{C}+1\mathbf{D}$

system and the affine coordinate system is given by $(X, Y, Z, T) \leftrightarrow (X/T, Y/Z^3)$, where $T = Z^2$. In ML-coordinates, the doubling, mixed addition and tripling formulae in projective coordinates require $5\mathbf{M}+3\mathbf{S}+3\mathbf{C}$, $8\mathbf{M}+2\mathbf{C}$ and $6\mathbf{M}+6\mathbf{C}$ respectively, where $\mathbf{S}$ denote the cost of a squaring in the finite field of characteristic three. It was noticed that a tripling algorithm cost $3\mathbf{M}+2\mathbf{S}+5\mathbf{C}+1\mathbf{D}$ using Jacobian projective coordinates in [17].

For convenience, we summarize all results into the following Table 1. From the table, we can see that the new proposed formulae are always more efficient than all previous formulae published for basic point operations on curves.

6 Conclusion

In this paper, new basic operation formulae are presented for Hessian curves over fields of characteristic 3. Also, new point representation called *scaled* projective is introduced for Weierstraß elliptic curves in characteristic three. Then, the efficient basic group operations are provided for this form.

We compared the performance of the proposed formulae to the previously best results for different coordinates systems. It is shown that the new formulae are superior to the previously known ones. It should be pointed out that, in double-base chain representation for a scalar number, the proposed point doubling and tripling may offer better performance.

References

1. Avanzi, R., Cohen, H., Doche, C., Frey, G., Lange, T., Nguyen, K., Vercauteren, F.: Handbook of Elliptic and Hyperelliptic Curve Cryptography. CRC Press (2005)
2. Blake, I.F., Seroussi, G., Smart, N.P.: Advances in Elliptic Curve Cryptography. Cambridge University Press (2005)
3. Blake, I.F., Seroussi, G., Smart, N.P.: Elliptic Curves in Cryptography, vol. 265. Cambridge University Press, New York (1999)

4. Bernstein, D.J., Kohel, D., Lange, T.: Twisted Hessian Curves, http://www.hyperelliptic.org/EFD/g1p/auto-twistedhessian.html
5. Chudnovsky, D.V., Chudnovsky, G.V.: Sequences of Numbers Generated by Addition in Formal Groups and New Primality and Factorization Tests. Advances in Applied Mathematics 7(4), 385–434 (1986)
6. Farashahi, R.R., Joye, M.: Efficient Arithmetic on Hessian Curves. In: Nguyen, P.Q., Pointcheval, D. (eds.) PKC 2010. LNCS, vol. 6056, pp. 243–260. Springer, Heidelberg (2010)
7. Fouquet, M., Gaudry, P., Harley, R.: An Extension of Satoh's Algorithm and its Implementation. J. Ramanujan Math. Soc. 15, 281–318 (2000)
8. Hankerson, D., Menezes, A.J., Vanstone, S.: Guide to Elliptic Curve Cryptography. Springer (2004)
9. Hesse, O.: Über die Elimination der Variabeln aus drei algebraischen Gleichungen vom zweiten Grade mit zwei Variabeln. Journal für Die Reine und Angewandte Mathematik 10, 68–96 (1844)
10. Hisil, H., Carter, G., Dawson, E.: New Formulae for Efficient Elliptic Curve Arithmetic. In: Srinathan, K., Rangan, C.P., Yung, M. (eds.) INDOCRYPT 2007. LNCS, vol. 4859, pp. 138–151. Springer, Heidelberg (2007)
11. Joye, M., Quisquater, J.-J.: Hessian Elliptic Curves and Side-Channel Attacks. In: Koç, Ç.K., Naccache, D., Paar, C. (eds.) CHES 2001. LNCS, vol. 2162, pp. 402–410. Springer, Heidelberg (2001)
12. Kim, K.H., Kim, S.I., Choe, J.S.: New Fast Algorithms for Arithmetic on Elliptic Curves over Fields of Characteristic Three. Cryptology ePrint Archive, Report 2007/179 (2007)
13. Koblitz, N.: Elliptic curve cryptosystems. Mathematics of Computation 48, 203–209 (1987)
14. Koblitz, N.: An Elliptic Curve Implementation of the Finite Field Digital Signature Algorithm. In: Krawczyk, H. (ed.) CRYPTO 1998. LNCS, vol. 1462, pp. 327–337. Springer, Heidelberg (1998)
15. López, J., Dahab, R.: Improved Algorithms for Elliptic Curve Arithmetic in tex2html_wrap_inline116. In: Tavares, S., Meijer, H. (eds.) SAC 1998. LNCS, vol. 1556, pp. 201–212. Springer, Heidelberg (1999)
16. Miller, V.S.: Use of Elliptic Curves in Cryptography. In: Williams, H.C. (ed.) CRYPTO 1985. LNCS, vol. 218, pp. 417–426. Springer, Heidelberg (1986)
17. Negre, C.: Scalar Multiplication on Elliptic Curves Defined over Fields of Small Odd Characteristic. In: Maitra, S., Veni Madhavan, C.E., Venkatesan, R. (eds.) INDOCRYPT 2005. LNCS, vol. 3797, pp. 389–402. Springer, Heidelberg (2005)
18. Satoh, T.: The canonical lift of an Ordinary Elliptic Curve over a Finite Field and its Point Counting. J. Ramanujan Math. Soc. 15, 247–270 (2000)
19. Smart, N.P.: The Hessian Form of an Elliptic Curve. In: Koç, Ç.K., Naccache, D., Paar, C. (eds.) CHES 2001. LNCS, vol. 2162, pp. 118–125. Springer, Heidelberg (2001)
20. Smart, N.P., Westwood, E.J.: Point Multiplication on Ordinary Elliptic Curves over Fields of Characteristic Three. Appl. Algebra Eng. Commun. Comput. 13(6), 485–497 (2003)

A Examples

In [20], Smart *et al.* provided an elliptic curve suitable for the current security level. According to the methods in [18,7], more ordinary curves over finite fields of characteristic three for high security level can be generated.

Here, we give some examples of ordinary elliptic curves over some finite field of characteristic three. The corresponding parameters are defined as follows.

m	The extension degree of the ternary field $\mathbb{F}_{3^m}$.
f	The reduction polynomial of degree m.
b	The parameter of the elliptic curve $\mathrm{E}: \; y^2 = x^3 + x^2 + b$.
r	The prime order of the main subgroup of $\mathrm{E}(\mathbb{F}_{3^m})$.
h	The cofactor, that is $h = \#\mathrm{E}(\mathbb{F}_{3^m})/r$.

E-151: $m = 151,\ f(z) = z^{151} + 2z^2 + 1,\ h = 3$ $b =$ 0x1FC4865AFE00A9216B0B5FD32C6300C4BED0707AE4072A03E55299F157B; $r =$ 0x359BA2B98CA11D6864A331B45AE711875640BA8E1297230F9EB217FB8393.
E-181: $m = 181,\ f(z) = z^{181} + 2z^{37} + 1,\ h = 3$ $b =$ 0x173CB756670960FD06D9438C9A55BE469574A995718B1786C9DAD40C45A7 AC68C208FC3; $r =$ 0x27367561CDDFD3AAFB8EA1FD4470B1171C349B993B5282BC17E661A1B1 DF65BCE845A035.
E-263: $m = 263,\ f(z) = z^{263} + 2z^{69} + 1,\ h = 3$ $b =$ 0x1E47D9F0855EB0ADDCE5948A2A1E5AF24EBFCC3051D647877CFFB91F5 64568C5103A09F22B234CE422567E0629358A740B8944C; $r =$ 0x994BBF51A32F5E702E4A3FFB7539AC6AAEAAF9B49E4CCA1DE8CE23F9 79DDA476F721963D0BF18B1216F037A8877236007190FD2F.
E-331: $m = 331,\ f(z) = z^{331} + 2z^2 + 1,\ h = 3$ $b =$ 0x52056E6E1C557FC37DD4D21EFFE1D5CA8E1528695E4B13536CF990AE79 C9242B8602535C92522A4EBB87E522ABF5C1CEA952EE52B9F6EA7389304 02CA3713AA0; $r =$ 0x8361D3334042B3F713BEB5D2C7BFAE83C436C40B479A21A4D1BE815079 F3C07FF992C36206C4E5B5DC9C2206CFB7F1AC1BD0F98A64CAB13DB5 3403AC4007E4875E5.
E-337: $m = 337,\ f(z) = z^{337} + 2z^3 + 1,\ h = 3$ $b =$ 0x359059FA58F98216D63B1FA12F4C194A09FDCFAF27CEEC308FB55B26938 D4A1D2E73ED6E9A17CDF7A84D1FAEDB14E38FC212CD76E460C3C5BFF 688234724B3EC0921; $r =$ 0x17621926CF1FDF27A973A13C53AD0D7F539BFF4441EE5E9CE59477E3E2B 471F2C6735F0933BB1C1B7ECA1A64D72D8F8F9336B4EE7CCA98AE54623C 8C15D6EF02AC7395.

Efficient Implementation of Bilinear Pairings on ARM Processors

Gurleen Grewal[1], Reza Azarderakhsh[1], Patrick Longa[2,⋆], Shi Hu[3], and David Jao[1]

[1] Dept. of Combinatorics & Optimization
University of Waterloo, Waterloo ON, N2L 3G1, Canada
grewal.gurleen@ymail.com,
{razarder,djao}@math.uwaterloo.ca
[2] Microsoft Research, 1 Microsoft Way, Redmond, WA 98052, USA
plonga@microsoft.com
[3] Stanford University, Stanford, California, USA
s3hu@stanford.edu

Abstract. As hardware capabilities increase, low-power devices such as smartphones represent a natural environment for the efficient implementation of cryptographic pairings. Few works in the literature have considered such platforms despite their growing importance in a post-PC world. In this paper, we investigate the efficient computation of the Optimal-Ate pairing over Barreto-Naehrig curves in software at different security levels on ARM processors. We exploit state-of-the-art techniques and propose new optimizations to speed up the computation in the tower field and curve arithmetic. In particular, we extend the concept of lazy reduction to inversion in extension fields, analyze an efficient alternative for the sparse multiplication used inside the Miller's algorithm and reduce further the cost of point/line evaluation formulas in affine and projective homogeneous coordinates. In addition, we study the efficiency of using M-type sextic twists in the pairing computation and carry out a detailed comparison between affine and projective coordinate systems. Our implementations on various mass-market smartphones and tablets significantly improve the state-of-the-art of pairing computation on ARM-powered devices, outperforming by at least a factor of 3.7 the best previous results in the literature.

Keywords: Optimal-Ate pairing, Barreto-Naehrig curves, ARM processor, pairing implementation.

1 Introduction

In the past decade, bilinear pairings have found a range of constructive applications in areas such as identity-based encryption and short signatures. Naturally, implementing such protocols requires efficient computation of the pairing function. Considerable work has been done to compute fast pairings on PCs [3,6,8,15,17]. Most

⋆ This work was partially carried out while this author was a postdoctoral researcher in the Department of Combinatorics and Optimization at the University of Waterloo.

recently, Aranha et al. [3] have computed the O-Ate pairing at the 128-bit security level in under 2 million cycles on various 64-bit PC processors. In contrast, relatively few articles [1,10] have considered efficient software implementations of pairings on ARM-based platforms such as hand-held smartphones and tablets. These platforms are widely predicted to become a dominant computing platform in the near future. Therefore, efficient implementation of pairing-based protocols for these devices is crucial for deployment of pairing-based cryptography in a mobile world and represents a natural area of research.

In this paper, we investigate efficient pairing computations at multiple security levels across different generations of ARM-based processors. We extend the work of Aranha et al. [3] to different BN curves and higher security levels. In addition, we make several further optimizations and analyze different options available for implementation at various stages of the pairing computation. We summarize our contributions as follows:

- Firstly, we extend the concept of lazy reduction employed by Aranha et al. [3] (see also Longa [13, Chapter 6]) to inversion in extension fields. We also optimize the sparse multiplication algorithm in the degree 12 extension.
- We examine different choices of towers for extension field arithmetic over various prime fields including BN-254, BN-446, and BN-638 [8]. We determine the most efficient implementation of extension fields in the context of pairing computation over BN-curves from the various choices available.
- The M-type sextic twist [16] has been largely ignored for use in pairing computations, most likely due to the inefficient untwisting map. We demonstrate that by computing the pairing on the twisted curve, we can bypass the inefficient untwisting. As a result, for the purposes of optimization one can use either M-type or D-type twists, thus roughly doubling the available choice of curves.
- Finally, we implement the proposed algorithms for computing the O-Ate pairing over BN curves on different ARM-based platforms and compare our measured timing results to their counterparts in the literature. Our experimental results are 3 to 5 times faster than the fastest available in prior literature, depending on the security level.

Acar et al. [1] have recently raised the question of whether affine coordinates or projective coordinates are a better choice for curve arithmetic in the context of software implementation of pairings. Their conclusion is that affine coordinates are faster at all security levels at or above 128 bits on the ARM platform. In contrast, our results (Section 6.2) demonstrate a clear advantage for homogeneous projective coordinates at the 128-bit security level, although affine coordinates remain faster at the 256-bit security level. We believe that our findings are more reliable since they represent a more realistic amount of optimization of the underlying field arithmetic implementation. We stress that, except for the work described in Section 6.1, our code does not contain any hand-optimized assembly or any overly aggressive optimizations that would compromise portability or maintainability.

The rest of this paper is organized as follows. In Section 2, we provide some background on the O-Ate pairing. In Section 3, we discuss the representation of extension fields. In Section 4, we describe arithmetic on BN curves including point addition and doubling presented in different coordinates. We provide operation counts for our algorithms in Section 5. In Section 6, we present the results of our implementation of the proposed scheme for computing O-Ate pairings on different ARM processors, and compare them with prior work.

2 Preliminaries

Barreto and Naehrig [4] describe a family of pairing friendly curves $E : y^2 = x^3 + b$ of order n with embedding degree 12 defined over a prime field $\mathbb{F}_q$ where q and n are given by the polynomials:

$$\begin{aligned} q &= 36x^4 + 36x^3 + 24x^2 + 6x + 1 \\ n &= 36x^4 + 36x^3 + 18x^2 + 6x + 1, \end{aligned} \tag{1}$$

for some integer x such that both q and n are prime and $b \in \mathbb{F}_q^*$ such that $b+1$ is a quadratic residue.

Let $\Pi_q : E \rightarrow E$ be the q-power Frobenius. Set $\mathbb{G}_1 = E[n] \cap \ker(\Pi_q - [1])$ and $\mathbb{G}_2 = E[n] \cap \ker(\Pi_q - [q])$. It is known that points in $\mathbb{G}_1$ have coordinates in $\mathbb{F}_q$, and points in $\mathbb{G}_2$ have coordinates in $\mathbb{F}_{q^{12}}$. The Optimal-Ate or O-Ate pairing [18] on E is defined by:

$$a_{\text{opt}} : \mathbb{G}_2 \times \mathbb{G}_1 \rightarrow \mu_n, (Q, P) \rightarrow f_{6x+2,Q}(P) \cdot h(P) \tag{2}$$

where $h(P) = l_{[6x+2]Q,qQ}(P) l_{[6x+2]Q+qQ,-q^2Q}(P)$ and $f_{6x+2,Q}(P)$ is the appropriate Miller function. Also, $l_{Q_1,Q_2}(P)$ is the line arising in the addition of Q_1 and Q_2 at point P. This function can be computed using Miller's algorithm [14]. A modified version of the algorithm from [14] which uses a NAF representation of x is given in Algorithm 1.

Let ξ be a quadratic and cubic non-residue over $\mathbb{F}_{q^2}$. Then the curves $E' : y^2 = x^3 + \frac{b}{\xi}$ (D-type) and $E'' : y^2 = x^3 + b\xi$ (M-type) are sextic twists of E over $\mathbb{F}_{q^2}$, and exactly one of them has order dividing n [3]. For this twist, the image $\mathbb{G}_2'$ of $\mathbb{G}_2$ under the twisting isomorphism lies entirely in $E'(\mathbb{F}_{q^2})$. Instead of using a degree 12 extension, the point Q can now be represented using only elements in a quadratic extension field. In addition, when performing curve arithmetic and computing the line function in the Miller loop, one can perform the arithmetic in $\mathbb{G}_2'$ and then map the result to $\mathbb{G}_2$, which considerably speeds up operations in the Miller loop.

2.1 Notations and Definitions

Throughout this paper, lower case variables denote single-precision integers, upper case variables denote double-precision integers. The operation $\times$ represents

Algorithm 1. Miller's Algorithm for the O-Ate Pairing [14]

```
Input: Points $P, Q \in E[n]$ and integer $n = (n_{l-1}, n_{l-2}, \cdots, n_1, n_0)_2 \in \mathbb{N}$.
Output: $f_{n,P}(Q)^{\frac{q^k-1}{n}}$.
 1: $T \leftarrow P,\ f \leftarrow 1$
 2: for $i = l-2$ down to $0$ do
 3:     $f \leftarrow f^2 \cdot l_{T,T}(Q)$
 4:     $T \leftarrow 2T$
 5:     if $l_i \neq 0$ then
 6:         $f \leftarrow f \cdot l_{T,P}(Q)$
 7:         $T \leftarrow T + P$
 8:     end if
 9: end for
10: $f \leftarrow f^{\frac{q^k-1}{n}}$
11: return $f$
```

multiplication without reduction, and $\otimes$ represents multiplication with reduction. The quantities m, s, a, i, and r denote the times for multiplication, squaring, addition, inversion, and modular reduction in $\mathbb{F}_q$, respectively. Likewise, $\tilde{m}, \tilde{s}, \tilde{a}$, $\tilde{i}$, and $\tilde{r}$ denote times for multiplication, squaring, addition, inversion, and reduction in $\mathbb{F}_{q^2}$, respectively, and m_u, s_u, $\tilde{m}_u$, and $\tilde{s}_u$ denote times for multiplication and squaring without reduction in the corresponding fields. Finally, m_b, m_i, m_ξ, and m_v denote times for multiplication by the quantities b, i, ξ, and v from Section 3.

3 Representation of Extension Fields

Efficient implementation of the underlying extension fields is crucial to achieve fast pairing results. The IEEE P1363.3 standard [9] recommends using towers to represent $\mathbb{F}_{q^k}$. For primes q congruent to 3 mod 8, we employ the following construction of Benger and Scott [5] to construct tower fields:

Property 1. For approximately 2/3rds of the BN-primes $q \equiv 3 \bmod 8$, the polynomial $y^6 - \alpha$, $\alpha = 1 + \sqrt{-1}$ is irreducible over $\mathbb{F}_{q^2} = \mathbb{F}_q(\sqrt{-1})$.

This gives the following towering scheme:

$$\begin{cases} \mathbb{F}_{q^2} = \mathbb{F}_q[i]/(i^2 - \beta), & \text{where } \beta = -1. \\ \mathbb{F}_{q^6} = \mathbb{F}_{q^2}[v]/(v^3 - \xi), & \text{where } \xi = 1 + i. \\ \mathbb{F}_{q^{12}} = \mathbb{F}_{q^6}[w]/(w^2 - v). \end{cases}$$

Based on this scheme, multiplication by i requires one negation over $\mathbb{F}_q$, and multiplication by ξ requires only one addition over $\mathbb{F}_{q^2}$. For primes congruent to 7 mod 8, we use the following construction which can be proven using the same ideas as those in Benger and Scott [5]:

Property 2. For approximately 2/3rds of the BN-primes $q \equiv 7 \bmod 8$, the polynomial $y^6 - \alpha$, $\alpha = 1 + \sqrt{-2}$ is irreducible over $\mathbb{F}_{q^2} = \mathbb{F}_q(\sqrt{-2})$.

This gives the following towering scheme:

$$\begin{cases} \mathbb{F}_{q^2} = \mathbb{F}_q[i]/(i^2 - \beta), & \text{where } \beta = -2. \\ \mathbb{F}_{q^6} = \mathbb{F}_{q^2}[v]/(v^3 - \xi), & \text{where } \xi = 1 + i. \\ \mathbb{F}_{q^{12}} = \mathbb{F}_{q^6}[w]/(w^2 - v). \end{cases}$$

All things being equal, the towering scheme derived from Property 1 is slightly faster for a given bit size. However, in practice, desirable BN-curves are rare, and it is sometimes necessary to use primes $q \equiv 7 \bmod 8$ in order to optimize other aspects such as the Hamming weight of x. In particular, the curves BN-446 and BN-638 [8] have $q \equiv 7 \bmod 8$. In such cases, Property 1 does not apply, so we use the towering scheme derived from Property 2. We also considered other approaches to construct tower extensions as suggested in [8], but found the above schemes consistently resulted in faster pairings compared to the other options.

3.1 Finite Field Operations and Lazy Reduction

Aranha et al. [3] proposed a lazy reduction scheme for efficient pairing computation in tower-friendly fields and curve arithmetic using projective coordinates. We extensively exploit their method and extend it to field inversion and curve arithmetic over affine coordinates. The proposed schemes using lazy reduction for inversion are given in Algorithms 2, 3 and 4 for $\mathbb{F}_{q^2}$, $\mathbb{F}_{q^6}$ and $\mathbb{F}_{q^{12}}$, respectively. The total savings with lazy reduction vs. no lazy reduction are one $\mathbb{F}_q$-reduction in $\mathbb{F}_{q^2}$-inversion, and 36 $\mathbb{F}_q$-reductions in $\mathbb{F}_{q^{12}}$-inversion (improving upon [3] by 16 $\mathbb{F}_q$-reductions). Interestingly enough, if one applies the lazy reduction technique to the recent $\mathbb{F}_{q^{12}}$ inversion algorithm of Pereira et al. [8], it replaces two $\tilde{m}_u$ by two $\tilde{s}_u$ but requires five more $\tilde{r}$ operations, which ultimately makes it slower in practice in comparison with the proposed scheme.

Algorithm 2. Inversion over $\mathbb{F}_{q^2}$ employing lazy reduction technique

```
Input: $a = a_0 + a_1 i$; $a_0, a_1 \in \mathbb{F}_q$; $\beta$ is a quadratic non-residue over $\mathbb{F}_q$
Output: $c = a^{-1} \in \mathbb{F}_{q^2}$
  $T_0 \leftarrow a_0 \times a_0$
  $T_1 \leftarrow -\beta \cdot (a_1 \times a_1)$
  $T_0 \leftarrow T_0 + T_1$
  $t_0 \leftarrow T_0 \bmod p$
  $t_0 \leftarrow t_0^{-1} \bmod p$
  $c_0 \leftarrow a_0 \otimes t_0$
  $c_1 \leftarrow -(a_1 \otimes t_0)$
  return $c = c_0 + c_1 i$
```

Algorithm 3. Inversion over $\mathbb{F}_{q^6}$ employing lazy reduction technique

```
Input: $a = a_0 + a_1 v + a_2 v^2;\ a_0, a_1, a_2 \in \mathbb{F}_{q^2}$
Output: $c = a^{-1} \in \mathbb{F}_{q^6}$
  $T_0 \leftarrow a_0 \times a_0$
  $t_0 \leftarrow \xi a_1$
  $T_1 \leftarrow t_0 \times a_2$
  $T_0 \leftarrow T_0 - T_1$
  $t_1 \leftarrow T_0 \bmod p$
  $T_0 \leftarrow a_2 \times a_2$
  $T_0 \leftarrow \xi T_0$
  $T_1 \leftarrow a_0 \times a_1$
  $T_0 \leftarrow T_0 - T_1$
  $t_2 \leftarrow T_0 \bmod p$
  $T_0 \leftarrow a_1 \times a_1$
  $T_1 \leftarrow a_0 \times a_2$
  $T_0 \leftarrow T_0 - T_1$
  $t_3 \leftarrow T_0 \bmod p$
  $T_0 \leftarrow t_0 \times t_3$
  $T_1 \leftarrow a_0 \times t_1$
  $T_0 \leftarrow T_0 + T_1$
  $t_0 \leftarrow \xi a_2$
  $T_1 \leftarrow t_0 \times t_2$
  $T_0 \leftarrow T_0 + T_1$
  $t_0 \leftarrow T_0 \bmod p$
  $t_0 \leftarrow t_0^{-1}$
  $c_0 \leftarrow t_1 \otimes t_0$
  $c_1 \leftarrow t_2 \otimes t_0$
  $c_2 \leftarrow t_3 \otimes t_0$
  return $c = c_1 + c_2 v + c_3 v^2$
```

Algorithm 4. Inversion over $\mathbb{F}_{q^{12}}$ employing lazy reduction technique

```
Input: $a = a_0 + a_1 w;\ a_0, a_1 \in \mathbb{F}_{q^6}$
Output: $c = a^{-1} \in \mathbb{F}_{q^{12}}$
  $T_0 \leftarrow a_0 \times a_0$
  $T_1 \leftarrow v \cdot (a_1 \times a_1)$
  $T_0 \leftarrow T_0 - T_1$
  $t_0 \leftarrow T_0 \bmod p$
  $t_0 \leftarrow t_0^{-1} \bmod p$
  $c_0 \leftarrow a_0 \otimes t_0$
  $c_1 \leftarrow -a_1 \otimes t_0$
  return $c = c_0 + c_1 w$
```

The line function in the Miller loop evaluates to a sparse $\mathbb{F}_{q^{12}}$ element containing only three of the six basis elements over $\mathbb{F}_{q^2}$. Thus, when multiplying the line function output with $f_{i,Q}(P)$, one can utilize the sparseness property to avoid full $\mathbb{F}_{q^{12}}$ arithmetic (Algorithm 5). For the BN-254 curve [8], our sparse multiplication algorithm requires $13\tilde{m}$ and $44\tilde{a}$ when a D-type twist is involved.

Algorithm 5. D-type sparse-dense multiplication in $\mathbb{F}_{q^{12}}$

```
Input: $a = a_0 + a_1 w + a_2 vw;\ a_0, a_1, a_2 \in \mathbb{F}_{q^2}, b = b_0 + b_1 w;\ b_0, b_1 \in \mathbb{F}_{q^6}$
Output: $ab \in \mathbb{F}_{q^{12}}$
  $A_0 \leftarrow a_0 \times b_0[0],\ A_1 \leftarrow a_0 \times b_0[1],\ A_2 \leftarrow a_0 \times b_0[2]$
  $A \leftarrow A_0 + A_1 v + A_2 v^2$
  $B \leftarrow \text{Fq6SparseMul}(a_1 w + a_2 vw, b_1)$
  $c_0 \leftarrow a_0 + a_1, c_1 \leftarrow a_2, c_2 \leftarrow 0$
  $c \leftarrow c_0 + c_1 v + c_2 v^2$
  $d \leftarrow b_0 + b_1$
  $E \leftarrow \text{Fq6SparseMul}(c, d)$
  $F \leftarrow E - (A + B)$
  $G \leftarrow Bv$
  $H \leftarrow A + G$
  $c_0 \leftarrow H \bmod p$
  $c_1 \leftarrow F \bmod p$
  return $c = c_0 + c_1 w$
```

Algorithm 6. Fq6SparseMul, used in Algorithm 5

```
Input: $a = a_0 + a_1 v;\ a_0, a_1 \in \mathbb{F}_{q^2}, b = b_0 + b_1 v + b_2 v^2;\ b_0, b_1, b_2 \in \mathbb{F}_{q^2}$
Output: $ab \in \mathbb{F}_{q^6}$
  $A \leftarrow a_0 \times b_0,\ B \leftarrow a_1 \times b_1$
  $C \leftarrow a_1 \times b_2 \xi$
  $D \leftarrow A + C$
  $e \leftarrow a_0 + a_1, f \leftarrow b_0 + b_1$
  $E \leftarrow e \times f$
  $G \leftarrow E - (A + B)$
  $H \leftarrow a_0 \times b_2$
  $I \leftarrow H + B$
  return $D + Gv + Iv^2$
```

A similar dense-sparse multiplication algorithm works for M-type twists, and requires an extra multiplication by v. We note that our approach requires 13 fewer additions over $\mathbb{F}_{q^2}$ compared to the one used in [3] (lazy reduction versions).

3.2 Mapping from the Twisted Curve to the Original Curve

Suppose we take ξ (from the towering scheme) to be the cubic and quadratic non-residue used to generate the sextic twist of the BN-curve E. After manipulating points on the twisted curve, they need to be mapped to the original curve. In the case of a D-type twist, the untwisting isomorphism is given by:

$$\Psi : (x, y) \to (\xi^{\frac{1}{3}} x, \xi^{\frac{1}{2}} y) = (w^2 x, w^3 y), \tag{3}$$

where both w^2 and w^3 are basis elements, and hence the untwisting map is almost free. If one uses a M-type twist the untwisting isomorphism is given as follows:

$$\Psi : (x, y) \to (\xi^{-\frac{2}{3}} x, \xi^{-\frac{1}{2}} y) = (\xi^{-1} w^4 x, \xi^{-1} w^3 y). \tag{4}$$

Untwisting of (4) is not efficient as the one given in (3). However, if we compute the pairing value on the twisted curve instead of the original curve, then we do not need to use the untwisting map. Instead, we require the inverse map which is almost free. Therefore, we compute the pairing on the original curve E when a D-type twist is involved, and on the twisted curve E' when an M-type twist is involved. Using this approach, we have found that both twist types are equivalent in performance up to point/line evaluation. The advantage of being able to consider both twist types is the immediate availability of many more useful curves for pairing computation.

3.3 Final Exponentiation Scheme

We use the final exponentiation scheme proposed in [7], which represents the current state-of-the-art for BN curves. In this scheme, first $\frac{q^{12}-1}{n}$ is factored into q^6-1, q^2+1, and $\frac{q^4-q^2+1}{n}$. The first two factors are easy to exponentiate. The remaining exponentiation $\frac{q^4-q^2+1}{n}$ can be performed in the cyclotomic subgroup. Using the fact that any fixed non-degenerate power of a pairing is a pairing, we raise to a multiple of the remaining factor. Recall that q and n are polynomials in x, and hence so is the final factor. We denote this polynomial as $d(x)$. In [7] it is shown that

$$\begin{aligned}2x(6x^2+3x+1)d(x) = {} & \lambda_3 q^3 + \lambda_2 q^2 + \lambda_1 q + \lambda_0 1 + 6x + 12x^2 + 12x^3\\ & + (4x+6x^2+12x^3)p(x) + (6x+6x^2+12x^3)p(x)^2\\ & + (-1+4x+6x^2+12x^3)p(x)^3, \qquad (5)\end{aligned}$$

where

$$\begin{aligned}\lambda_3(x) &= -1+4x+6x^2+12x^3,\\ \lambda_2(x) &= 6x+6x^2+12x^3\\ \lambda_1(x) &= 4x+6x^2+12x^3\\ \lambda_0(x) &= 1+6x+12x^2+12x^3. \qquad (6)\end{aligned}$$

To compute (6), the following exponentiations are performed:

$$f \mapsto f^x \mapsto f^{2x} \mapsto f^{4x} \mapsto f^{6x} \mapsto f^{6x^2} \mapsto f^{12x^2} \mapsto f^{12x^3}. \qquad (7)$$

The cost of computing (7) is 3 exponentiations by x, 3 squarings and 1 multiplication. We then compute the terms $a = f^{12x^3} f^{6x^2} f^{6x}$ and $b = a(f^{2x})^{-1}$, which require 3 multiplications. The final pairing value is obtained as

$$af^{6x^2} f b^p a^{p^2} (bf^{-1})^{p^3}, \qquad (8)$$

which costs 6 multiplications and 6 Frobenius operations. In total, this method requires 3 exponentiations by x, 3 squarings, 10 multiplications, and 3 Frobenius operations. In comparison, the technique used in [3] requires 3 additional multiplications and an additional squaring, and thus is slightly slower.

4 Curve Arithmetic

In this section, we discuss our optimizations to curve arithmetic over affine and homogeneous projective coordinates. We also evaluated other coordinate systems such as Jacobian coordinates but found that none were faster than homogeneous coordinates for our application.

4.1 Affine Coordinates

Let the points $T = (x, y)$ and $Q = (x_2, y_2) \in E'(\mathbb{F}_q)$ be given in affine coordinates, and let $T + Q = (x_3, y_3)$ be the sum of the points T and Q. When $T = Q$ we have

$$\begin{aligned} m &= \frac{3x^2}{2y} \\ x_3 &= m^2 - 2x \\ y_3 &= (mx - y) - mx_3 \end{aligned}$$

For D-type twists, the secant or tangent line evaluated at $P = (x_P, y_P)$ is given by:

$$l_{2\Psi(T)}(P) = y_P - mx_P w + (mx - y)w^3. \tag{9}$$

To compute the above, we precompute $\bar{x}_P = -x_P$ (to save the cost of computing the negation on-the-fly) and use the following sequence of operations which requires $1\tilde{i}$, $3\tilde{m}$, $2\tilde{s}$, $7\tilde{a}$, and $2m$ if $T = Q$. In comparison, the doubling formula in Lauter et al. [12] costs 3 additional $\tilde{a}$.

$$A = \frac{1}{2y} \qquad B = 3x^2 \qquad C = AB \qquad D = 2x \qquad x_3 = C^2 - D$$
$$E = Cx - y \qquad y_3 = E - Cx_3 \qquad F = C\bar{x}_P$$
$$l_{2\Psi(T)}(P) = y_P + Fw + Ew^3$$

Similarly, when $T \neq Q$ we use the following sequence of operations which requires $1\tilde{i}$, $3\tilde{m}$, $1\tilde{s}$, $6\tilde{a}$, and $2m$ – saving $2\tilde{a}$ compared to the addition formula in Lauter et al. [12].

$$A = \frac{1}{y_2 - y} \qquad B = x_2 - x \qquad C = AB \qquad D = x + x_2 \qquad x_3 = C^2 - D$$
$$E = Cx - y \qquad y_3 = E - Cx_3 \qquad F = C\bar{x}_P$$
$$l_{2\Psi(T)}(P) = y_P + Fw + Ew^3$$

In an M-type twist, the tangent line evaluated at $\Psi(P) = (x_P w^2, y_P w^3)$ is given by:

$$l_{2T}(\Psi(P)) = y_P w^3 - mx_P w^2 + (mx - y), \tag{10}$$

and can be computed in a similar way.

4.2 Homogeneous Coordinates

During the first iteration of the Miller loop, the Z-coordinate of the point Q has value equal to 1. We use this fact to eliminate a multiplication and three squarings using a special first doubling routine in the first iteration. Recently, Arahna et al. [3], presented optimized formulas for point doubling/line evaluation. We note that the twisting point P given by $(x_P/w^2, y_P/w^3)$ is better represented by $(x_P w, y_P)$, which is obtained by multiplying by w^3. This eliminates the multiplication by ξ and gives the following revised formula. Let $T = (X, Y, Z) \in E'(\mathbb{F}_{q^2})$ be in homogeneous coordinates. Then $2T = (X_3, Y_3, Z_3)$ is given by:

$$\begin{aligned} X_3 &= \frac{XY}{2}(Y^2 - 9b'Z^2) \\ Y_3 &= \left[\frac{1}{2}(Y^2 + 9b'Z^2)\right]^2 - 27b'^2Z^4 \\ Z_3 &= 2Y^3Z \end{aligned}$$

In the case of a D-type twist, the corresponding line function evaluated at $P = (x_P, y_P)$ is given by:

$$l_{2\Psi(T)}(P) = -2YZy_P + 3X^2x_Pw + (3b'Z^2 - Y^2)w^3$$

We compute this value using the following sequence of operations.

$$A = \frac{XY}{2} \quad B = Y^2 \quad C = Z^2 \quad E = 3b'C \quad F = 3E \quad X_3 = A \cdot (B - F)$$

$$G = \frac{B + f}{2} \quad Y_3 = G^2 - 3E^2 \quad H = (Y + Z)^2 - (B + C) \quad Z_3 = B \cdot H$$

$$l_{2\Psi(T)}(P) = H\bar{y}_P + 3X^2x_Pw + (E - B)w^3$$

Aranha et al. [3] observe $\tilde{m} - \tilde{s} \approx 3\tilde{a}$ and hence computing XY directly is faster than using $(X + Y)^2$, Y^2 and X^2 on a PC. However, on ARM processors, we have $\tilde{m} - \tilde{s} \approx 6\tilde{a}$. Thus, the latter technique is more efficient on ARM processors. The overall cost of point doubling and line evaluation is $2\tilde{m}$, $7\tilde{s}$, $22\tilde{a}$, and $4m$, assuming that the cost of division by two and multiplication by b' are equivalent to the cost of addition. Similarly, we compute point addition and line function evaluation using the following sequence of operations which uses $11\tilde{m}$, $2\tilde{s}$, $8\tilde{a}$, and $4m$ (saving 2 $\mathbb{F}_{q^2}$ additions over [3]). Note that $\bar{x}_P$ and $\bar{y}_P$ are precomputed to save again the cost of computing $-x_P$ and $-y_P$.

$$A = Y_2Z \quad B = X_2Z \quad \theta = Y - A \quad \lambda = X - B \quad C = \theta^2$$

$$D = \lambda^2 \quad E = \lambda^3 \quad F = ZC \quad G = XD \quad H = E + F - 2G$$

$$X_3 = \lambda H \quad I = YE \quad Y_3 = \theta(G - H) - I \quad Z_3 = ZE \quad J = \theta X_2 - \lambda Y_2$$

$$l_{\Psi(T+Q)}(P) = \lambda\bar{y}_P + \theta\bar{x}_Pw + Jw^3$$

In the case of an M-type twist the corresponding line computation can be computed using the same sequences of operations as above. As in [3], we also use lazy reduction techniques to optimize the above formulae (see Table 1).

Table 1. Operation counts for 254-bit, 446-bit, and 638-bit prime fields

$E'(\mathbb{F}_{p^2})$ **Arith.**	254-bit	446-bit/638-bit
Doubl/Eval (Proj)	$2\tilde{m}_u + 7\tilde{s}_u + 8\tilde{r} + 25\tilde{a} + 4m$	$2\tilde{m}_u + 7\tilde{s}_u + 8\tilde{r} + 34\tilde{a} + a + 4m$
Doubl/Eval (Affi)	$\tilde{i} + 3\tilde{m}_u + 2\tilde{s}_u + 5\tilde{r} + 7\tilde{a} + 2m$	$\tilde{i} + 3\tilde{m}_u + 2\tilde{s}_u + 5\tilde{r} + 7\tilde{a} + 2m$
Add./Eval (Pro)	$11\tilde{m}_u + 2\tilde{s}_u + 11\tilde{r} + 10\tilde{a} + 4m$	$11\tilde{m}_u + 2\tilde{s}_u + 11\tilde{r} + 10\tilde{a} + 4m$
Add/Eval (Affi)	$\tilde{i} + 3\tilde{m}_u + \tilde{s}_u + 4\tilde{r} + 6\tilde{a} + 2m$	$\tilde{i} + 2\tilde{m}_u + \tilde{s}_u + 3\tilde{r} + 6\tilde{a} + 2m$
First doubl./Eval	$3\tilde{m}_u + 4\tilde{s}_u + 7\tilde{r} + 14\tilde{a} + 4m$	$3\tilde{m}_u + 4\tilde{s}_u + 7\tilde{r} + 23\tilde{a} + a + 4m$
p-power Frob.	$2\tilde{m} + 2a$	$8\tilde{m} + 2a$
p^2- power Frob.	$4m$	$16\tilde{m} + 4a$

$\mathbb{F}_{p^2}$ **Arith.**	254-bit	446-bit/638-bit
Add/Subtr./Nega.	$\tilde{a} = 2a$	$\tilde{a} = 2a$
Mult.	$\tilde{m} = \tilde{m}_u + \tilde{r} = 3m_u + 2r + 8a$	$\tilde{m} = \tilde{m}_u + \tilde{r} = 3m_u + 2r + 10a$
Squaring	$\tilde{s} = \tilde{s}_u + \tilde{r} = 2m_u + 2r + 3a$	$\tilde{s} = \tilde{s}_u + \tilde{r} = 2m_u + 2r + 5a$
Mult. by β	$m_b = a$	$m_b = 2a$
Mult. by ξ	$m_\xi = 2a$	$m_\xi = 3a$
Inversion	$\tilde{i} = i + 2m_u + 2s_u + 3r + 3a$	$\tilde{i} = i + 2m_u + 2s_u + 3r + 5a$

$\mathbb{F}_{p^{12}}$ **Arith.**	254-bit	446-bit/638-bit
Multi.	$18\tilde{m}_u + 110\tilde{a} + 6\tilde{r}$	$18\tilde{m}_u + 117\tilde{a} + 6\tilde{r}$
Sparse Mult.	$13\tilde{m}_u + 6\tilde{r} + 48\tilde{a}$	$13\tilde{m}_u + 6\tilde{r} + 54\tilde{a}$
Sparser Mult.	$6\tilde{m}_u + 6\tilde{r} + 13\tilde{a}$	$6\tilde{m}_u + 6\tilde{r} + 14\tilde{a}$
Affi. Sparse Mult.	$10\tilde{m}_u + 6\tilde{r} + 47\tilde{a} + 6m_u + a$	$10\tilde{m}_u + 53\tilde{a} + 6\tilde{r} + 6m_u + a$
Squaring	$12\tilde{m}_u + 6\tilde{r} + 73\tilde{a}$	$12\tilde{m}_u + 6\tilde{r} + 78\tilde{a}$
Cyclotomic Sqr.	$9\tilde{s}_u + 46\tilde{a} + 6\tilde{r}$	$9\tilde{s}_u + 49\tilde{a} + a + 6\tilde{r}$
Simult. Decomp.	$9\tilde{m} + 6\tilde{s} + 22\tilde{a} + \tilde{i}$	$9\tilde{m} + 6\tilde{s} + 24\tilde{a} + \tilde{i}$ (BN-446) $16\tilde{m} + 9\tilde{s} + 35\tilde{a} + \tilde{i}$ (BN-638)
p-power Frob.	$5\tilde{m} + 6a$	$5\tilde{m} + 6a$
p^2-power Frob.	$10m + 2\tilde{a}$	$10m + 2\tilde{a}$
Expon. by x	$45\tilde{m}_u + 378\tilde{s}_u +$ $275\tilde{r} + 2164\tilde{a} + \tilde{i}$	$45\tilde{m}_u + 666\tilde{s}_u + 467\tilde{r}_u +$ $3943\tilde{a} + \tilde{i}$ (BN-446)
		$70\tilde{m} + 948\tilde{s} + 675\tilde{r} +$ $5606\tilde{a} + 158a + \tilde{i}$ (BN-638)
Inversion	$25\tilde{m}_u + 9\tilde{s}_u + 16\tilde{r} + 121\tilde{a} + \tilde{i}$	$25\tilde{m}_u + 9\tilde{s}_u + 18\tilde{r} + 138\tilde{a} + \tilde{i}$
Compressed Sqr.	$6\tilde{s}_u + 31\tilde{a} + 4\tilde{r}$	$6\tilde{s}_u + 33\tilde{a} + a + 4\tilde{r}$

5 Operation Counts

We provide here detailed operation counts for our algorithms on the BN-254, BN-446, and BN-638 curves used in Acar et. al [1] and defined in [8]. Table 1 provides the operation counts for all component operations. Numbers for BN-446 and BN-638 are the same except where indicated.

For BN-254, using the techniques described above, the projective pairing Miller loop executes one negation in $\mathbb{F}_q$, one first doubling with line evaluation, 63 point doublings with line evaluations, 6 point additions with line evaluations, one p-power Frobenius in $E'(\mathbb{F}_{p^2})$, one p^2-power Frobenius in $E'(\mathbb{F}_{p^2})$, 66

Table 2. Cost of the computation of O-Ate pairings using various coordinates

Curve	Coordinates	Cost
BN-254	Proj. Miller loop	$1841\tilde{m}_u + 457\tilde{s}_u + 1371\tilde{r} + 9516\tilde{a} + 284m + 3a$
	Aff. Miller loop	$70\tilde{i} + 1658\tilde{m}_u + 134\tilde{s}_u + 942\tilde{r} + 8292\tilde{a} + 540m + 132a$
	Final exp.	$386\tilde{m}_u + 1164\tilde{s}_u + 943\tilde{r} + 4\tilde{i} + 7989\tilde{a} + 30m + 15a$
BN-446	Proj. Miller loop	$3151\tilde{m}_u + 793\tilde{s}_u + 2345\tilde{r} + 18595\tilde{a} + 472m + 117a$
	Aff. Miller loop	$118\tilde{i} + 2872\tilde{m}_u + 230\tilde{s}_u + 1610\tilde{r} + 15612\tilde{a} + 920m + 230a$
	Final exp.	$386\tilde{m}_u + 2034\tilde{s}_u + 1519\tilde{r} + 4\tilde{i} + 13374\tilde{a} + 30m + 345a$
BN-638	Proj. Miller loop	$4548\tilde{m}_u + 1140\tilde{s}_u + 3557\tilde{r} + 27198\tilde{a} + 676m + 166a$
	Aff. Miller loop	$169\tilde{i} + 4143\tilde{m}_u + 330\tilde{s}_u + 2324\tilde{r} + 22574\tilde{a} + 1340m + 333a$
	Final exp.	$436\tilde{m}_u + 2880\tilde{s}_u + 2143\tilde{r} + 4\tilde{i} + 18528\tilde{a} + 30m + 489a$

sparse multiplications, 63 squarings in $\mathbb{F}_{p^2}$, 1 negation in $E'(\mathbb{F}_{p^2})$, 2 sparser (i.e. sparse-sparse) multiplications [3], and 1 multiplication in $\mathbb{F}_{p^{12}}$. Using Table 1, we compute the total number of operations required in the Miller loop using homogeneous projective coordinates to be

$$\begin{aligned}\text{ML254P} &= a + 3\tilde{m}_u + 7\tilde{r} + 14\tilde{a} + 4m + 63(2\tilde{m}_u + 7\tilde{s}_u + 8\tilde{r} + 25\tilde{a} + 4m) + \\ &\quad 6(11\tilde{m}_u + 2\tilde{s}_u + 11\tilde{r} + 10\tilde{a} + 4m) + 2\tilde{m} + 2a + 4m + \\ &\quad 66(\tilde{m}_u + 6\tilde{r} + 48\tilde{a}) + \ 63(12\tilde{m}_u + 6\tilde{r} + 73\tilde{a}) + \tilde{a} + \\ &\quad 2(6\tilde{m}_u + 6\tilde{r} + 13\tilde{a}) + 18\tilde{m}_u + 110\tilde{a} + 6\tilde{r} \\ &= 1841\tilde{m}_u + 457\tilde{s}_u + 1371\tilde{r} + 9516\tilde{a} + 284m + 3a.\end{aligned}$$

Similarly, we also compute the Miller loop operation costs for BN-446 and BN-638 and for projective and affine coordinates, and give the results in Table 2.

We also compute the operation count for the final exponentiation. For BN-254, the final exponentiation requires 6 conjugations in $\mathbb{F}_{p^{12}}$, one negation in $E'(\mathbb{F}_{p^2})$, one inversion in $\mathbb{F}_{p^{12}}$, 12 multiplications in $\mathbb{F}_{p^{12}}$, two p-power Frobenius in $\mathbb{F}_{p^{12}}$, 3 p^2-power Frobenius in $\mathbb{F}_{p^{12}}$, 3 exponentiations by x, and 3 cyclotomic squarings. Based on these costs, we compute the total number of operations required in the final exponentiation and give the result in Table 2. Similarly, we also compute the final exponentiation operation cost for BN-446 and BN-638. The total operation count for the pairing computation is the cost of the Miller loop plus the final exponentiation.

6 Implementation Results

To evaluate the performance of the proposed schemes for computing the O-Ate pairing in practice, we implemented them on various ARM processors. We used the following platforms in our experiments.

– A Marvell Kirkwood 6281 ARMv5 CPU processor (Feroceon 88FR131) operating at 1.2 GHz. In terms of registers it has 16 32-bit registers $\texttt{r}_0$ to $\texttt{r}_{15}$ of which two are for the stack pointer and program counter, leaving only 14 32-bit registers for general use.

- An iPad 2 (Apple A5) using an ARMv7 Cortex-A9 MPCore processor operating at 1.0 GHz clock frequency. It has 16 128-bit vector registers which are available as 32 64-bit vector registers, as these registers share physical space with the 128-bit vector registers.
- A Samsung Galaxy Nexus (1.2 GHz TI OMAP 4460 ARM Cortex-A9). The CPU microarchitecture is identical to the Apple A5. We included it to examine whether different implementations of the Cortex-A9 core have comparable performance in this application.

Our software is based on version 0.2.3 of the RELIC toolkit [2], with the GMP 5.0.2 backend, modified to include our optimizations. Except for the work described in Section 6.1, all of our software is platform-independent C code, and the same source package runs unmodified on all the above ARM platforms as well as both x86 and x86-64 Linux and Windows PCs. Our implementation also supports and includes BN curves at additional security levels beyond the three presented here. For each platform, we used the standard operating system and development environment that ships with the device, namely Debian Squeeze (native C compiler), XCode 4.3.0, and Android NDK (r7c) for the Kirkwood, iPad, and Galaxy Nexus respectively.

We present the results of our experiments in Table 3. For ease of comparison we have also included the numbers from [1] in Table 3. Roughly speaking, our timings are over three times faster than the results appearing in [1], which itself represents the fastest reported times prior to our work. Specifically, examining our iPad results, which were obtained on an identical micro-architecture and clock speed, we find that our implementation is 3.7, 3.7, and 5.4 times faster on BN-254, BN-446, and BN-638, respectively. Some, but not all, of the improvement can be attributed to faster field arithmetic; for example, $\mathbb{F}_q$-field multiplication on the RELIC toolkit is roughly 1.4 times as fast on the iPad 2 compared to [1]. A more detailed comparison based on operation counts is difficult because [1] does not provide any operation counts, and also because our strategy and our operation counts rely on lazy reduction, which does not play a role in [1].

6.1 Assembly Optimization

In order to investigate the potential performance gains available from hand optimized machine code, we implemented the two most commonly used field arithmetic operations (addition and multiplication) for the BN-254 curve in ARM assembly instructions. Due to the curve-specific and platform-specific nature of this endeavor, we performed this work only for the BN-254 curve and only on the Linux platforms (Marvell Kirkwood and Galaxy Nexus).

The main advantage of assembly language is that it provides more control for lower level arithmetic computations. Although the available C compilers are quite good, they still produce inefficient code since in the C language it is infeasible to express instruction priorities. Moreover, one can use hand-optimized assembly code to decompose larger computations into small pieces suitable for vectorization. We employ the following techniques to optimize our implementation in assembly:

Table 3. Timings for affine and projective pairings on different ARM processors and comparisons with prior literature. Times for the Miller loop (ML) in each row reflect those of the faster pairing.

Marvell Kirkwood (ARM v5) Feroceon 88FR131 at 1.2 GHz [This work]														
Field Size	Lang	Operation Timing [μs]												
		a	m	r	i	I/M	$\tilde{a}$	$\tilde{m}$	$\tilde{s}$	$\tilde{i}$	ML	FE	O-A(a)	O-A(p)
254-bit	ASM	0.12	1.49	1.12	17.53	11.8	0.28	4.08	3.44	23.57	9,722	6,176	16,076	15,898
	C	0.18	1.74	1.02	17.40	10.0	0.35	4.96	4.01	24.01	11,877	7,550	19,427	19,509
446-bit		0.20	3.79	2.25	34.67	9.1	0.38	10.74	8.57	48.90	42,857	23,137	65,994	65,958
638-bit		0.27	6.82	3.83	52.33	7.7	0.51	18.23	14.93	77.11	98,044	51,351	149,395	153,713

iPad 2 (ARM v7) Apple A5 Cortex-A9 at 1.0 GHz [This work]														
Field Size	Lang	Operation Timing [μs]												
		a	m	r	i	I/M	$\tilde{a}$	$\tilde{m}$	$\tilde{s}$	$\tilde{i}$	ML	FE	O-A(a)	O-A(p)
254-bit	C	0.16	1.28	0.93	13.44	10.5	0.25	3.48	2.88	19.19	8,338	5,483	14,604	13,821
446-bit		0.16	2.92	1.62	27.15	9.3	0.26	8.03	6.46	37.95	32,087	17,180	49,365	49,267
638-bit		0.20	5.58	2.92	43.62	7.8	0.34	15.07	12.09	64.68	79,056	40,572	119,628	123,410

Galaxy Nexus (ARM v7) TI OMAP 4460 Cortex-A9 at 1.2 GHz [This work]														
Field Size	Lang	Operation Timing [μs]												
		a	m	r	i	I/M	$\tilde{a}$	$\tilde{m}$	$\tilde{s}$	$\tilde{i}$	ML	FE	O-A(a)	O-A(p)
254-bit	ASM	0.05	0.93	0.55	9.42	10.1	0.10	2.46	2.07	13.79	6,147	3,758	10,573	9,905
	C	0.07	0.98	0.53	9.62	9.8	0.13	2.81	2.11	14.05	6,859	4,382	11,839	11,241
446-bit		0.12	2.36	1.27	23.08	9.8	0.22	6.29	5.17	32.27	25,792	13,752	39,886	39,544
638-bit		0.19	4.87	3.05	38.45	7.9	0.45	12.20	10.39	56.78	65,698	33,658	99,356	99,466

NVidia Tegra 2 (ARM v7) Cortex-A9 at 1.0 GHz [1]														
Field Size	Lang	Operation Timing [μs]												
		a	m	r	i	I/M	$\tilde{a}$	$\tilde{m}$	$\tilde{s}$	$\tilde{i}$	ML	FE	O-A(a)	O-A(p)
254-bit	C	0.67	1.72	n/a	18.35	10.7	1.42	8.18	5.20	26.61	26,320	24,690	51,010	55,190
446-bit		1.17	4.01	n/a	35.85	8.9	2.37	17.24	10.84	54.23	97,530	86,750	184,280	195,560
638-bit		1.71	8.22	n/a	56.09	6.8	3.48	31.81	20.55	91.92	236,480	413,370	649,850	768,060

– Loop unrolling: since the maximum number of bits of the operands are known, it makes sense to unroll all loops in order to provide us the ability to avoid conditional branches (which basically eliminates branch prediction misses in the pipeline), reorder the instructions, and insert carry propagate codes at desired points.
– Instruction re-ordering: by careful reordering of non-dependent instructions (in terms of data and processing units), it is possible to minimize the number of pipeline stalls and therefore execute the code faster. Two of the most frequent multi-cycle instructions used in our code are word multiplication and memory reads. Each 32-bit word multiplication takes between 3 and 6 cycles and each memory read needs 2 cycles. By applying loop unrolling, it is possible to load the data required for the next multiplication while the pipeline is performing the current multiplication. Also, lots of register clean-ups and carry propagation codes are performed while the pipeline is doing a multiplication.
– Register allocation: all of the available registers were used extensively in order to eliminate the need to access memory for fetching the operands or store partial results. This improves overall performance considerably.
– Multiple stores: ARM processors are capable of loading and storing multiple words from or to the memory by one instruction. By storing the final result at once instead of writing a word back to memory each time when a new result is ready, we minimize the number of memory access instructions.

Also, we do some register clean-ups (cost-free) when the pipeline is performing the multiple store instruction. It is worth mentioning that while it was possible to write 8 words at once, only 4 words are written to memory at each time because the available non-dependent instructions to re-order after the multiple store instruction are limited.

Table 3 includes our measurements of pairing computation times using our assembly implementation alongside the results for the C implementation. We find that the BN-254 pairing using hand-optimized assembly code is roughly 20% faster than the C implementation. In all cases, the projective pairing benefits more than the affine pairing, because we did not hand-optimize the inversion routine in assembly.

6.2 Affine vs. Homogeneous Coordinates

Acar et al. [1] assert that on ARM processors, small inversion to multiplication (I/M) ratios over $\mathbb{F}_q$ render it more efficient to compute a pairing using affine coordinates. If we are using a prime q congruent to $3 \bmod 8$, then compared to a projective doubling step, an affine doubling step costs an extra $\tilde{i}$ and an unreduced multiplication, and saves $5\tilde{s}_u + 3\tilde{r} + 16.5\tilde{a} + 2m$. Compared to a first doubling, it costs an extra $\tilde{i}$ and saves $2\tilde{s}_u + 2\tilde{r} + 6.5\tilde{a} + 2m$. An addition step costs an extra $\tilde{i}$ and saves $8\tilde{m}_u + \tilde{s}_u + 7\tilde{r} + 3\tilde{a} + 2m$; and a dense-sparse multiplication needs an additional $6m$ and saves $3\tilde{m}_u + 3\tilde{r} + 0.5\tilde{a}$. Thus, the difference between an affine and projective pairing is $70i - 659m_u - 396r - 4417a$.

From Table 3, we observe that the two pairings are roughly equal in performance at about the 446-bit field size. At this field size, we have $m \approx 1.5r$ and $m \approx 15a$. Plugging these estimates into the expression $70i - 659m_u - 396r - 4417a$, we find that an affine pairing is expected to be faster than a projective pairing whenever the I/M ratio in the base field falls below about 10.0. The results of Table 3 indicate that our I/M ratios cross this point slightly above 446 bits. We observe that both affine and projective pairings achieve similar performance in our implementation on ARM processors, with an advantage in the range of 1%-6% for projective coordinates on BN-254, and a slight advantage for affine coordinates on BN-638, with slightly better results for projective coordinates on BN-446. These results overturn those of Acar et al. [1] which show too much advantage in favor of affine coordinates (well above 5%). However, there are situations in which affine coordinates would be preferable even at the 128-bit security level (e.g., products of pairings). Different conclusions may hold for assembly-optimized variants at higher security levels, which are not included in our analysis.

7 Conclusions

In this paper, we present high speed implementation results of the Optimal-Ate pairing on BN curves for different security levels. We extend the concept of lazy reduction to inversion in extension fields and optimize the sparse multiplication algorithm in the degree 12 extension. Our work indicates that D-type and M-type twists achieve equivalent performance for point/line evaluation computation, with only a very slight advantage in favor of D-type when computing sparse multiplications. In addition, we include an efficient method from [7] to perform final exponentiation and reduce its computation time. Finally, we measure the Optimal-Ate pairing over BN curves on different ARM-based platforms and compare the timing results to the leading ones available in the open literature. Our timing results are over three times faster than the previous fastest results appearing in [1]. Although the authors in [1] find affine coordinates to be faster on ARM in all cases, based on our measurements we conclude that homogeneous projective coordinates are unambiguously faster than affine coordinates for O-Ate pairings at the 128-bit security level when higher levels of optimization are used.

Acknowledgments. The authors would like to thank the reviewers for their comments. This work has been supported by NSERC CRD Grant CRDPJ 405857-10 and an Ontario Ministry of Research and Innovation Early Researcher Award.

References

1. Acar, T., Lauter, K., Naehrig, M., Shumow, D.: Affine Pairings on ARM. IACR Cryptology ePrint Archive, 2011:243 (2011), Pairing 2012 (to appear)
2. Aranha, D.F., Gouvêa, C.P.L.: RELIC is an Efficient LIbrary for Cryptography, `http://code.google.com/p/relic-toolkit/`
3. Aranha, D.F., Karabina, K., Longa, P., Gebotys, C.H., López, J.: Faster Explicit Formulas for Computing Pairings over Ordinary Curves. In: Paterson, K.G. (ed.) EUROCRYPT 2011. LNCS, vol. 6632, pp. 48–68. Springer, Heidelberg (2011)
4. Barreto, P.S.L.M., Naehrig, M.: Pairing-Friendly Elliptic Curves of Prime Order. In: Preneel, B., Tavares, S. (eds.) SAC 2005. LNCS, vol. 3897, pp. 319–331. Springer, Heidelberg (2006)
5. Benger, N., Scott, M.: Constructing Tower Extensions of Finite Fields for Implementation of Pairing-Based Cryptography. In: Hasan, M.A., Helleseth, T. (eds.) WAIFI 2010. LNCS, vol. 6087, pp. 180–195. Springer, Heidelberg (2010)
6. Beuchat, J.-L., González-Díaz, J.E., Mitsunari, S., Okamoto, E., Rodríguez-Henríquez, F., Teruya, T.: High-speed software implementation of the optimal ate pairing over Barreto-Naehrig curves. In: Joye, et al. (eds.) [11], pp. 21–39
7. Fuentes-Castañeda, L., Knapp, E., Rodríguez-Henríquez, F.: Faster Hashing to $\mathbb{G}_2$. In: Miri, A., Vaudenay, S. (eds.) SAC 2011. LNCS, vol. 7118, pp. 412–430. Springer, Heidelberg (2012)
8. Pereira Geovandro, C.C.F., Simplício Jr., M.A., Naehrig, M., Barreto, P.S.L.M.: A family of implementation-friendly BN elliptic curves. Journal of Systems and Software 84(8), 1319–1326 (2011)
9. IEEE Std. 1363-3/D1. Draft Standard for Identity-based public key cryptography using pairings (January 2008)
10. Iyama, T., Kiyomoto, S., Fukushima, K., Tanaka, T., Takagi, T.: Efficient Implementation of Pairing on BREW Mobile Phones. In: Echizen, I., Kunihiro, N., Sasaki, R. (eds.) IWSEC 2010. LNCS, vol. 6434, pp. 326–336. Springer, Heidelberg (2010)

11. Joye, M., Miyaji, A., Otsuka, A. (eds.): Pairing 2010. LNCS, vol. 6487. Springer, Heidelberg (2010)
12. Lauter, K., Montgomery, P.L., Naehrig, M.: An analysis of affine coordinates for pairing computation. In: Joye, et al. (eds.) [11], pp. 1–20
13. Longa, P.: High-Speed Elliptic Curve and Pairing-Based Cryptography. PhD thesis, University of Waterloo (April 2011), http://hdl.handle.net/10012/5857
14. Miller, V.S.: The Weil pairing, and its efficient calculation. J. Cryptology 17(4), 235–261 (2004)
15. Naehrig, M., Niederhagen, R., Schwabe, P.: New Software Speed Records for Cryptographic Pairings. In: Abdalla, M., Barreto, P.S.L.M. (eds.) LATINCRYPT 2010. LNCS, vol. 6212, pp. 109–123. Springer, Heidelberg (2010)
16. Scott, M.: A note on twists for pairing friendly curves. Personal webpage (2009), ftp://ftp.computing.dcu.ie/pub/resources/crypto/twists.pdf
17. Scott, M.: On the Efficient Implementation of Pairing-Based Protocols. In: Chen, L. (ed.) IMACC 2011. LNCS, vol. 7089, pp. 296–308. Springer, Heidelberg (2011)
18. Vercauteren, F.: Optimal pairings. IEEE Transactions on Information Theory 56(1), 455–461 (2010)

Towards Faster and Greener Cryptoprocessor for Eta Pairing on Supersingular Elliptic Curve over $\mathbb{F}_{2^{1223}}$

Jithra Adikari[1], M. Anwar Hasan[2], and Christophe Negre[3,4,5]

[1] Elliptic Technologies Inc., Ottawa ON, Canada
[2] Department of Electrical and Computer Engineering, University of Waterloo, Canada
[3] Team DALI, Université de Perpignan, France
[4] LIRMM, UMR 5506, Université Montpellier 2, France
[5] LIRMM, UMR 5506, CNRS, France

Abstract. At the CHES workshop last year, Ghosh *et al.* presented an FPGA based cryptoprocessor, which for the first time ever makes it possible to compute an eta pairing at the 128-bit security level in less than one milli-second. The high performance of their cryptoprocessor comes largely from the use of the Karatsuba method for field multiplication. In this article, for the same type of pairing we propose hybrid sequential/parallel multipliers based on the Toeplitz matrix-vector products and present some optimizations for the final exponentiation, resulting in high performance cryptoprocessors. On the same kind of FPGA devices, our cryptoprocessor performs pairing faster than that of [12] while requiring less hardware resources. We also present ASIC implementations and report that the three-way split multiplier based cryptoprocessor consumes less energy than the two-way.

1 Introduction

Since 2001, cryptographic *pairing* has been used extensively to develop various security protocols, including the well known identity based encryption [3] and the short signature scheme [4]. For such protocols, pairing is by far the most computation intensive operation. A pairing algorithm typically requires thousands of additions and multiplications followed by a final exponentiation over very large finite fields. From the implementation point of view, pairing is thus very challenging; in fact it is computationally far more demanding than classical cryptographic schemes such as elliptic curve cryptography.

In this paper, we consider hardware implementation of a type of pairing known as the η_T pairing [17] on elliptic curves defined over extended binary fields. Specifically, we focus on the 128-bit security level. During the past few years several pairing implementations for 128-bit security level have been published for various field characteristics [13,6,7,12,1,15,10,11]. Here we consider pairing over elliptic curve $E(\mathbb{F}_{2^{1223}})$, for which we also need to deal computations over $\mathbb{F}_{2^{4\cdot 1223}}$. In CHES 2011, Ghosh *et al.* [12] have proposed a cryptoprocessor architecture for computing such η_T pairing at the 128-bit security level, and reported its implementation results based on field programmable gate arrays (FPGA). The cryptoprocessor has a hybrid sequential/parallel architecture for multiplication in $\mathbb{F}_{2^{1223}}$ and performs the inversion of a non-zero element of $\mathbb{F}_{2^{4 \cdot 1223}}$ using linear algebra. More specifically, using the Karatsuba formula, a multiplication in $\mathbb{F}_{2^{1223}}$ is broken down into nine separate multiplications of polynomials of size 306 bits each. This allows the cryptoprocessor perform one $\mathbb{F}_{2^{1223}}$ multiplication in ten clock cycles, i.e., nine cycles are used for nine 306-bit multiplications and one cycle for the reconstruction and the reduction of the product. For Xilinx Virtex6 FPGA, the cryptoprocessor of Ghosh *et al.* [12] takes 190 μs only, making it the fastest 128-bit secure η_T pairing unit available up until now (also see [1] for a more recent comparison).

Our Work: In this paper, we propose a new cryptoprocessor for the 128-bit security level η_T pairing on the same supersingular elliptic curve used in [12]. The proposed cryptoprocessor is different than that in [12] in a number of ways, and when implemented on the same type of FPGA devices, it performs the pairing in much less time. The primary difference, which is also the main source of improvements, lies in the multiplier over $\mathbb{F}_{2^{1223}}$, which is typically the most area consuming component of such a cryptoprocessor. We use an asymptotically better, namely Toeplitz-matrix vector product (TMVP) based approach for multiplication in $\mathbb{F}_{2^{1223}}$. To the best of our knowledge, this is the first time that TMVP based multipliers are used in the implementation of pairing. The two-way split and the three-way split TMVP formulas of [8] result in multipliers which are more efficient in area and time compared to those based on the corresponding Karatsuba formulas. For example, the two-way TMVP formula enables us perform one $\mathbb{F}_{2^{1223}}$ multiplication in nine clock cycles (instead of ten cycles using the Karatsuba), and the three-way formula does it only in six clock cycles without a proportional increase in area.

In our work, we also improve the final exponentiation operation for the pairing cryptoprocessor. Typically, this exponentiation is performed via costly operations including several multiplications over $\mathbb{F}_{2^{1223}}$ along with an inversion and an exponentiation to the power of 2^{612} over $\mathbb{F}_{2^{4 \cdot 1223}}$. We reduce the complexity of the inversion by adapting the norm based approach [5] over the tower fields $\mathbb{F}_{2^{1223}} \subset \mathbb{F}_{2^{2 \cdot 1223}} \subset \mathbb{F}_{2^{4 \cdot 1223}}$ We find that a square root can take a considerably fewer number of bit operations than a squaring operation in $\mathbb{F}_{2^{1223}}$, and following [2], we use this feature to reduce the computational cost of the exponentiation to the power 2^{612} by replacing the sequence of squaring by a sequence of square root operations.

In terms of hardware realization, we report both FPGA and application specific integrated circuit (ASIC) implementations of the proposed cryptoprocessor. To the best of our knowledge, these are the first ASIC implementations for η_T pairing at the 128-bit security level using binary supersingular curves. Based on the ASIC results, we find that the three-way split multiplier based cryptoprocessor is a greener choice as it consumes less energy than its two-way counterpart.

Organization: The remainder of this paper is organized as follows: in Section 2 we briefly review η_T pairing and analyze different operations involved in it. In Section 3 we briefly review two approaches to multiply elements of $\mathbb{F}_{2^{1223}}$ based on Toeplitz-matrix vector products and provide their respective implementation results. Then in Section 4, we present several improvements to the final exponentiation in η_T pairing. In Section 5 we present the overall architecture for the η_T pairing and implementation results. We compare our schemes with best known results and wind up the paper with some concluding remarks in Section 6.

2 Pairing Algorithm

Pairing: We consider supersingular elliptic curve E defined by the following equation $Y^2 + Y = X^3 + X$ over the finite field $\mathbb{F}_q$, where $q = 2^{1223}$. As stated in [17], we have $\#E(\mathbb{F}_{2^{1223}}) = 5r$ where $r = (2^{1223}+2^{612}+1)/5$ is a 1221-bit prime divisor and the curve E has an embedding degree $k = 4$. We construct the field $\mathbb{F}_{q^k} = \mathbb{F}_{2^{4\cdot1223}}$ through two extensions of degree two $\mathbb{F}_{2^{2\cdot1223}} = \mathbb{F}_{2^{1223}}[u]/(u^2 + u + 1)$ and $\mathbb{F}_{2^{4\cdot1223}} = \mathbb{F}_{2^{2\cdot1223}}[u]/(v^2 + v + u)$. Now if we denote μ_r the subgroup of order r of $\mathbb{F}^*_{q^k} = \mathbb{F}^*_{2^{4\cdot1223}}$ and if we pick an element $P \in E(\mathbb{F}_{2^{1223}})$ of order r, we can define the η_T pairing

$$\eta_T \colon \langle P\rangle \times \langle P\rangle \longrightarrow \mu_r$$

as $\eta_T(P_1, P_2) = e(P_1, \psi(P_2))$ where e is the Tate paring and $\psi(x, y) = (x + u^2, y + xu + v)$. The security level of this pairing is equal to 128 bits. In [17] the authors have proposed Algorithm 1 for the computation of the η_T pairing.

Algorithm 1. η_T pairing [17]

```
Require: $P_2 = (x_1, y_1)$ and $P_2 = (x_2, y_2) \in E(\mathbb{F}_{2^{1223}})[r]$
Ensure: $\eta_T(P_1, P_2)$
  $T \leftarrow x_1 + 1$
  $f \leftarrow T \cdot (x_1 + x_2 + 1) + y_1 + y_2 + (T + x_2)u + v$
  for $i = 1$ to 612 do
    $T \leftarrow x_1$, $x_1 \leftarrow \sqrt{x_1}$, $y_1 \leftarrow \sqrt{y_1}$
    $g \leftarrow T \cdot (x_1 + x_2) + y_1 + y_2 + x_1 + 1 + (T + x_2)u + v$
    $f \leftarrow f \cdot g$
    $x_2 \leftarrow x_2^2$, $y_2 \leftarrow y_2^2$
  end for
  return($f^{(2^{2\cdot1223}-1)(2^{1223}-2^{612}+1)}$)
```

The **for** loop in Algorithm 1 is a re-expression of the Miller's loop of the Tate pairing for the special curve E and the η_T pairing considered here. We remark that the main operations performed in Algorithm 1 are two square roots in $\mathbb{F}_{2^{1223}}$ in the first step of the **for** loop, one multiplication $T \cdot (x_1 + x_2)$ in $\mathbb{F}_{2^{1223}}$ plus several additions for the computation of g, one special multiplication

$f \cdot g$ in $\mathbb{F}_{2^{4\cdot 1223}}$ for the computation of f, two squarings in $\mathbb{F}_{2^{1223}}$ and the final exponentiation $f^{(2^{2\cdot 1223}-1)(2^{1223}-2^{612}+1)}$ in $\mathbb{F}_{2^{4\cdot 1223}}$.

Addition of two elements of a binary field corresponds to bit-wise XOR operations and, for field $\mathbb{F}_{2^{1223}}$, the bit-parallel implementation of an adder requires 1223 two-input XOR gates. Below we briefly describe squaring and square root operations for elements of $\mathbb{F}_{2^{1223}}$. The other operations are discussed in subsequent sections.

Square and Square Root: The field $\mathbb{F}_{2^{1223}}$ is constructed as $\mathbb{F}_{2^{1223}} = \mathbb{F}_2[x]/(x^{1223} + x^{255} + 1)$. The squaring of $A = \sum_{i=0}^{1223} a_i x^i$ in $\mathbb{F}_{2^{1223}}$ can, in this situation, be performed as follows

$$\begin{aligned} A^2 &= \textstyle\sum_{i=0}^{1222} a_i x^{2i} \mod (x^{1223} + x^{255} + 1) \\ &= \textstyle\sum_{i=0}^{127} a_i x^{2i} + \sum_{i=128}^{254} (a_i + a_{i+612-128} + a_{i+1224-256}) x^{2i} \\ &\quad + \textstyle\sum_{i=255}^{611} (a_i + a_{i+612-128}) x^{2i} + \sum_{i=0}^{126} (a_{i+612} + a_{i+1224-128}) x^{2i+1} \\ &\quad + \textstyle\sum_{i=127}^{610} a_{i+612} x^{2i+1}. \end{aligned}$$

Based on the aforementioned expression, the squaring in $\mathbb{F}_{2^{1223}}$ can be implemented with 738 XOR gates and a delay of $2D_X$.

The square root of an element $A = \sum_{i=0}^{1223} a_i x^i$ in $\mathbb{F}_{2^{1223}}$ can be expressed as

$$\begin{aligned} \sqrt{A} &= \sqrt{\textstyle\sum_{i=0}^{611} a_{2i} x^{2i}} + \sqrt{\textstyle\sum_{i=0}^{610} a_{2i+1} x^{2i+1}} \\ &= \left(\textstyle\sum_{i=0}^{611} a_{2i} x^i\right) + \sqrt{x}\left(\textstyle\sum_{i=0}^{610} a_{2i+1} x^i\right). \end{aligned}$$

Since $x = x^{256} + x^{1224} \mod (1 + x^{255} + x^{1223})$, we have $\sqrt{x} = x^{128} + x^{612} \mod (1 + x^{255} + x^{1223})$, and, after replacing $\sqrt{x}$ with $x^{256} + x^{1224}$, we obtain that $\sqrt{A}$ can be computed as

$$\begin{aligned} \sqrt{A} &= \textstyle\sum_{i=0}^{127} a_{2i} x^i + \sum_{i=128}^{611} (a_{2i} + a_{2i-256+1}) x^i \\ &\quad + \textstyle\sum_{i=0}^{126} (a_{2i+1} + a_{2(i+612-128)+1}) x^{i+612} + \sum_{i=127}^{610} a_{2i+1} x^{i+612}. \end{aligned}$$

Hence a square root can be implemented with 611 XOR gates and a delay of D_X. Consequently, the number of bit operations is lower in a square root than squaring in $\mathbb{F}_{2^{1223}}$ defined by $x^{1223} + x^{255} + 1$. We take advantage of this feature in the final exponentiation of pairing.

Unlike squaring and square root operations, the various multiplications appearing in the η_T paring algorithm are more difficult to implement. In the next section we present two multiplier architectures used in our proposal for η_T pairing cryptoprocessor.

3 Multiplier Architectures

In the η_T pairing algorithm, the main operations include multiplications with inputs in one of the three fields $\mathbb{F}_{2^{1223}} = \mathbb{F}_2[x]/(x^{1223} + x^{255} + 1)$, $\mathbb{F}_{2^{2\cdot 1223}} = \mathbb{F}_{2^{1223}}[u]/(u^2 + u + 1)$ and $\mathbb{F}_{2^{4\cdot 1223}} = \mathbb{F}_{2^{2\cdot 1223}}[v]/(v^2 + v + u)$. The authors in [12] have designed a multiplier architecture for $\mathbb{F}_{2^{1223}}$ and perform multiplications in the extended fields $\mathbb{F}_{2^{2\cdot 1223}}$ and $\mathbb{F}_{2^{4\cdot 1223}}$ through a sequence of multiplications in $\mathbb{F}_{2^{1223}}$. This method for the multiplication over $\mathbb{F}_{2^{2\cdot 1223}}$ and $\mathbb{F}_{2^{4\cdot 1223}}$ is reviewed later in Subsection 3.3. For the multiplication in $\mathbb{F}_{2^{1223}}$ the authors in [12] use a hybrid sequential and parallel recursion of the Karatsuba formula for polynomial multiplication. In the next subsection we investigate an alternative approach, which is based on the formulation of multiplication over $\mathbb{F}_{2^{1223}}$ as Toeplitz matrix vector products.

3.1 Multiplication in $\mathbb{F}_{2^{1223}}$ through Toeplitz Matrix Vector Products

The field $\mathbb{F}_{2^{1223}}$ is the set of binary polynomials of degree < 1223 modulo the irreducible trinomial $x^{1223}+x^{255}+1$. For two given elements $A = \sum_{i=0}^{1222} a_i x^i$ and $B = \sum_{i=0}^{1222} b_i x^i$, the product of A and B can be computed by first performing a polynomial multiplication and then reduce it modulo $x^{1223}+x^{255}+1$. As stated in [16], this multiplication and reduction can be re-expressed as follows

$$\begin{aligned} A \times B \mod (x^{1223}+x^{255}+1) &= A \times (\textstyle\sum_{i=0}^{1222} b_i x^i) \mod (x^{1223}+x^{255}+1) \\ &= \textstyle\sum_{i=0}^{1222} A^{(i)} b_i \end{aligned}$$

where $A^{(i)} = (x^i \times A) \mod (x^{1223}+x^{255}+1)$. This means that the product in $\mathbb{F}_{2^{1223}}$ can be seen as a matrix-vector product $M_A \times B$ where $M_A = \left[A^{(0)}\ A^{(1)}\ \cdots\ A^{(1222)}\right]$. We can further arrange this matrix-vector product. Indeed if we define the following 1223×1223 circulant matrix

$$U = \begin{bmatrix} 0 & I_{968\times 968} \\ I_{255\times 255} & 0 \end{bmatrix},$$

then matrix $T_A = U \cdot M_A$ is obtained by removing the top 255 rows and placing them below the other 968 rows of M_A. The resulting matrix T_A has the following Toeplitz structure:

$$\begin{bmatrix}
a_{255} & a_{254}+a_{1222} & \cdots & a_0+a_{968} & a_{1222}+a_{967} & \cdots & a_{510}+a_{255} & \cdots & a_{257}+a_2+a_{970} & a_{256}+a_1+a_{969} \\
a_{256} & a_{255} & \cdots & a_1+a_{969} & a_0+a_{968} & \cdots & a_{511}+a_{256} & \cdots & a_{258}+a_3+a_{971} & a_{257}+a_2+a_{970} \\
\vdots & & & & & & & & & \vdots \\
a_{1221} & a_{1220} & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & a_{1222}+a_{967} \\
a_{1222} & a_{1221} & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & a_0+a_{968} \\
a_0 & a_{1222} & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & a_1+a_{969} \\
\vdots & & & & & & & & & \vdots \\
a_{253} & a_{252} & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & a_{254}+a_{1222} \\
a_{254} & a_{253} & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & a_{255}
\end{bmatrix}.$$

The product $C = A \times B \mod (x^{1223}+x^{255}+1)$ is obtained by first performing this Toeplitz matrix vector product $C' = T_A \cdot B$ and then by switching the top 968 coefficients of C' with its other 255 coefficients.

We take advantage of the Toeplitz matrix vector product (TMVP) expression of a multiplication in $\mathbb{F}_{2^{1223}}$ since a TMVP can be performed using a subquadratic method [19,8]. This subquadratic method is obtained by recursively applying two-way or three-way split formulas. Assuming that T is an $n \times n$ Toeplitz matrix and V is a column vector with n rows, the two-way split formula reduces a TMVP of size n to three TMVPs of size $n/2$ each as follows:

$$T \cdot V = \begin{bmatrix} T_1 & T_0 \\ T_2 & T_1 \end{bmatrix} \cdot \begin{bmatrix} V_0 \\ V_1 \end{bmatrix} = \begin{bmatrix} P_0 + P_1 \\ P_2 + P_1 \end{bmatrix}, \text{ where } \begin{cases} P_0 = (T_0 + T_1) \cdot V_1, \\ P_1 = T_1 \cdot (V_0 + V_1), \\ P_2 = (T_1 + T_2) \cdot V_0. \end{cases} \tag{1}$$

In [8] we can also find a three-way split formula which reduces a TMVP of size n to six TMVPs of size $n/3$ as follows:

$$T \cdot V = \begin{bmatrix} T_2 & T_1 & T_0 \\ T_3 & T_2 & T_1 \\ T_4 & T_3 & T_2 \end{bmatrix} \cdot \begin{bmatrix} V_0 \\ V_1 \\ V_2 \end{bmatrix} = \begin{bmatrix} P_0 + P_3 + P_4 \\ P_1 + P_3 + P_5 \\ P_2 + P_4 + P_5 \end{bmatrix}, \text{ where } \begin{cases} P_0 = (T_0 + T_1 + T_2) \cdot V_2, \\ P_1 = (T_1 + T_2 + T_3) \cdot V_1, \\ P_2 = (T_2 + T_3 + T_4) \cdot V_0, \\ P_3 = T_1 \cdot (V_1 + V_2), \\ P_4 = T_2 \cdot (V_0 + V_2), \\ P_5 = T_3 \cdot (V_0 + V_1). \end{cases} \tag{2}$$

When applied recursively, the above formulas provide a multiplication with subquadratic complexity.The complexities of the two- and three-way split formulas are reported in Table 1. For details on the evaluation of these complexities the reader may refer to [8]. For comparison purposes, we also provide the complexities of the two-way and the three-way polynomial approaches presented in [18] and [9]. The complexities in Table 1 show that, in each type of splits, TMVP approaches outperform polynomial approaches and are thus more suitable to design finite field multipliers.

Table 1. Area and time complexities of two-way and three-way split polynomial multiplication and TMVP

Formula type	Method	#AND	#XOR	Delay
Two-way	Poly. mult. with Karatsuba ([18])	$n^{\log_2(3)}$	$6n^{\log_2(3)} - 8n + 2$	$D_A + 3\log_2(n)D_X$
	Poly. mult. with [9]	$n^{\log_2(3)}$	$6n^{\log_2(3)} - 8n + 2$	$D_A + 2\log_2(n)D_X$
	TMVP with [8]	$n^{\log_2(3)}$	$5.5n^{\log_2(3)} - 7n + 1.5$	$D_A + 2\log_2(n)D_X$
Three-way	Poly. mult. with [18]	$n^{\log_3(6)}$	$5.33n^{\log_3(6)} - 7.33n + 2$	$D_A + 4\log_3(n)D_X$
	Poly. mult. with [9]	$n^{\log_3(6)}$	$5.33n^{\log_3(6)} - 7.33n + 2$	$D_A + 3\log_3(n)D_X$
	TMVP with [8]	$n^{\log_3(6)}$	$4.8n^{\log_3(6)} - 5n + 0.2$	$D_A + 3\log_3(n)D_X$

3.2 Hybrid Sequential/Parallel TMVP Multiplier

In order to use the TMVP formulas (1) and (2) in our design of a multiplier for the field $\mathbb{F}_{2^{1223}}$, we begin with two important issues. First, in order to apply the TMVP formula, the size of the considered TMVP must be divisible by 2 or 3. The solution we adopt here is to extend the matrices (by extending the diagonal) and the vectors (by padding 0) up to a size divisible by 2 or 3. For example for $n = 1223$ we can extend it to 1224, which is divisible by 2 and 3. If needed, we can repeat this strategy during the recursion.

The second and more challenging issue is that a fully parallel multiplier for the field $\mathbb{F}_{2^{1223}}$ is too large to fit in common FPGA devices. To this end, we adopt the approach used in [12] in the case of polynomial multiplication with the Karatsuba formula. Specifically, instead of performing all the computations in parallel, we process the recursion of the TMVP formulas through a hybrid sequential/parallel process. Below we describe this approach by applying *one* recursion of the three-way split formula. We have done also an hybrid sequential/parallel multiplier based on *two* recursions of the two-way split TMVP formula (1). Because of lack of space, we cannot present this case in details in this paper. The reader may refer to the forthcomming technical report extending the current paper for these details.

Three-Way Hybrid Sequential/parallel TMVP Multiplier for $n = 1224$. The first hybrid sequential/parallel TMVP multiplier for $n = 1224$ includes hardware for splitting the associated Toeplitz matrices and vectors of size 1224 in three ways. The splitting leads to six TMVP instances of size 408 each. One fully parallel hardware unit performs these six TMVPs one after another. Some additional hardware is used to temporarily store and combine the resulting 408-bit outputs into a 1224-bit product.

The overall architecture of this multiplier is depicted in Fig. 1 and a brief operational description follows.

Referring to Fig. 1, the entries of the Toeplitz matrix and the vector are stored it two registers, namely T_{in} and R_{in}. We note that the Toeplitz matrix is completely defined by its top row and left most column. Following (2), the entries are split as follows:

$$T \cdot V = \begin{bmatrix} T_2 & T_1 & T_0 \\ T_3 & T_2 & T_1 \\ T_4 & T_3 & T_2 \end{bmatrix} \cdot \begin{bmatrix} V_0 \\ V_1 \\ V_2 \end{bmatrix}$$

where matrices $T_i, i = 0, 1, \ldots, 4$ and vectors $V_i, i = 0, 1, 2$ are of size 408. The *CONVERTOR* block computes, through from the 1223 bit stored in T_{in} the coefficients of the first column and the first row of T. Then the T_{net} block of Fig. 1 sequentially generates the six matrices

$$(T_2 + T_1 + T_0),\ (T_3 + T_2 + T_1),\ (T_4 + T_3 + T_2),\ T_1,\ T_2,\ T_3,$$

involved in the six products $P_i, i = 0, \ldots, 5$ of (2). In parallel to the formation of the matrices, the V_{net} block of Fig. 1 generates the following six corresponding vectors

$$V_2,\ V_1,\ V_0,\ (V_1 + V_2),\ (V_0 + V_2),\ (V_0 + V_1).$$

The (matrix and vector) outputs of T_{net} and V_{net} are input to a fully parallel unit which computes TMVP of size 408 each. The parallel unit consists of three

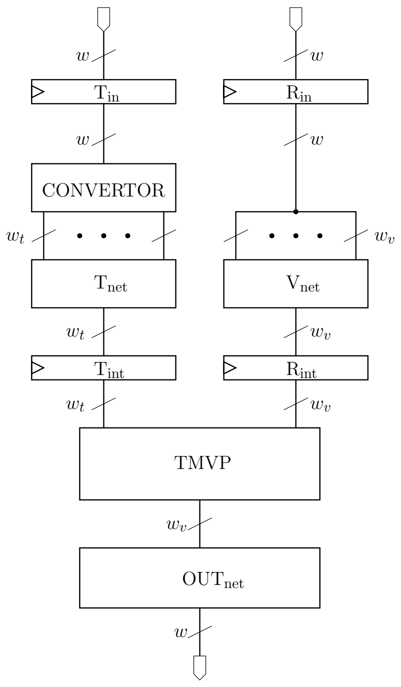


Fig. 1. Multiplier architecture

recursions of three-way split TMVP formula and one two-way split TMVP formula: $408 \to 136 \to 46 \to 16 \to 8$, the TMVPs of size 8 are performed with a quadratic method.

This parallel unit sequentially outputs the six products $P_0, P_1, \ldots, P_5$ defined in (2). These products $P_i, i = 0, \ldots, 5$, are accumulated in the block labeled as OUT_{net} of Fig. 1 to form the result $W = T \cdot V$. Some additional details of blocks T_{net}, V_{net} and OUT_{net} are depicted in Fig. 2.

Implementation Results of the Multiplier. We have implemented in FPGA and ASIC the two-way and three-way hybrid sequential/parallel multipliers presented in the previous subsections. Our main goal has been to optimize the speed. Below we present the FPGA implementations; for lack of space we omit the ASIC implementations for the multipliers, but we do report the ASIC implementations of the complete cryptoprocessor later in the paper.

The multiplier designs have been placed and routed on Xilinx `xc6vlx365t-3` FPGA by executing Xilinx Integrated Software Environment (ISE$^{\mathrm{TM}}$) version `12.4`. The synthesis tool for FPGA estimated LUT counts and operating frequencies for our three- and two-way hybrid sequential/parallel multipliers. In Table 2 we have reported these results along with the results of Ghosh *et al.* [12].

Table 2 shows that the proposed two-way hybrid sequential/parallel multiplier is better than the multiplier presented in [12], considering both LUTs and

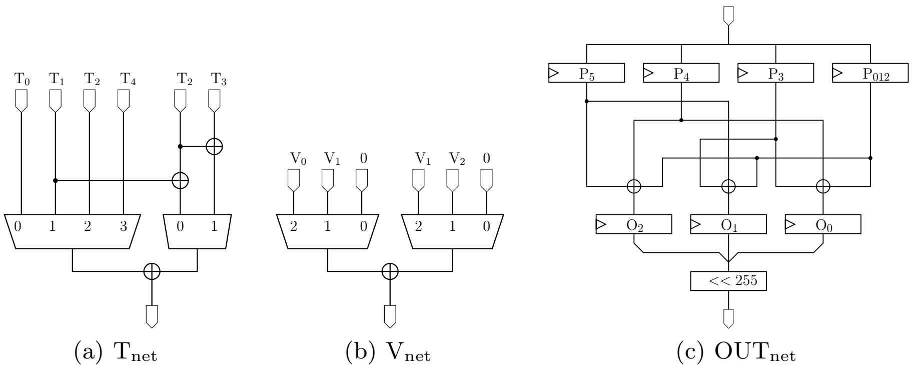


Fig. 2. Blocks of the hybrid three-way sequential/parallel multiplier

Table 2. FPGA Place & Route implementation synthesis results and comparisons for $\mathbb{F}_{2^{1223}}$ multipliers

Multiplier architecture	LUTs	Freq. (MHz)	# Clock Cycles per mult	Latency (ns)	Area × time (LUTs) (ms)
Sequential use of 306 bit parallel Karatsuba Mult [12]	30,148	250	10	40.0	1.21
Two-way 306-bit TMVP Mult	19,721	271	9	33.2	0.65
Three-way 408-bit TMVP Mult (Subsec. 3.2)	33,546	267	6	22.5	0.75

frequency. The improvement in the number of LUT is mainly due to the use of TMVP approach for finite field multiplication. The reduction of the delay is partly explained by the use of the TMVP approach, but is also due to the reduced number of clock cycle (9 instead of 10) needed for a multiplication. The proposed three-way hybrid sequential/parallel multiplier present an alternative to the proposed two-way multiplier and the multiplier of [12]. Specifically, it has the largest number of LUTs, but offers the highest frequency and smallest serial use number (i.e., 6 vs. 9 and 10). This results in a multiplier which is less area efficient but faster than the proposed two-way multiplier.

3.3 Multiplication in Fields $\mathbb{F}_{2^{2\cdot1223}}$ and $\mathbb{F}_{2^{4\cdot1223}}$

In this section we review the method used to multiply two elements in $\mathbb{F}_{2^{2\cdot1223}}$ and to multiply two elements in $\mathbb{F}_{2^{4\cdot1223}}$. This method uses one recursion (resp. two recursions) of Karatsuba to reduce the multiplication in $\mathbb{F}_{2^{2\cdot1223}}$ (resp. $\mathbb{F}_{2^{4\cdot1223}}$) to several multiplications in $\mathbb{F}_{2^{1223}}$. These multiplications are performed through one of the two multipliers presented in the previous subsection.

Multiplication in $\mathbb{F}_{2^{2\cdot 1223}}$*:* The elements of $\mathbb{F}_{2^{2\cdot 1223}} = \mathbb{F}_{2^{1223}}[u]/(u^2+u+1)$ are considered as degree one polynomial in u. We can thus reduce one multiplication in $\mathbb{F}_{2^{2\cdot 1223}}$ to three multiplications in $\mathbb{F}_{2^{1223}}$ using the Karatsuba formula. Indeed, let $A = A_0 + A_1 u$ and $B = B_0 + B_1 u$ be two elements of $\mathbb{F}_{2^{2\cdot 1223}}$. We perform these three products $P_0 = A_0 B_0$, $P_1 = (A_0 + A_1)(B_0 + B_1)$ and $P_2 = A_1 B_1$. We then reconstruct and reduce modulo u^2+u+1 the product $C = A \times B$ as follows

$$\begin{aligned} C &= P_0 + (P_1 + P_0 + P_2)u + P_2 u^2 \\ &= P_0 + P_2 + (P_1 + P_0)u \mod (u^2+u+1) \end{aligned}$$

The cost of this method is equal to 3 multiplications and 4 additions in the field $\mathbb{F}_{2^{1223}}$.

Multiplication in $\mathbb{F}_{2^{4\cdot 1223}}$*.* The elements of $\mathbb{F}_{2^{4\cdot 1223}} = \mathbb{F}_{2^{2\cdot 1223}}[v]/(v^2+v+u)$ are considered as degree one polynomial in v. We reduce one multiplication in $\mathbb{F}_{2^{4\cdot 1223}}$ to three multiplications in $\mathbb{F}_{2^{2\cdot 1223}}$ by applying the Karatsuba formula. Indeed, let $A = (A_0 + A_1 u) + (A_2 + A_3 u)v$ and $B = B_0 + B_1 u + (B_2 + B_3 u)v$ be two elements of $\mathbb{F}_{2^{4\cdot 1223}}$. We first perform three products $P_0 + P_1 u = (A_0 + A_1 u)(B_0 + B_1 u), P_2 + P_3 u = (A_0 + A_2 + (A_1 + A_3)u)(B_0 + B_2 + (B_1 + B_3)u)$ and $P_4 + P_5 u = (A_2 + A_3 u)(B_2 + B_3 u)$ in $\mathbb{F}_{2^{2\cdot 1223}}$ and then reconstruct $C = A \times B$ modulo $v^2 + v + u$ as follows

$$\begin{aligned} C &= (P_0 + P_1 u) + (P_2 + P_3 u + P_0 + P_1 u + P_4 + P_5 u)v + (P_4 + P_5 u)v^2 \\ &= (P_0 + P_5 + (P_1 + P_4 + P_5)u) \\ &\quad + (P_2 + P_0 + (P_3 + P_1)u)v \mod (v^2+v+u, u^2+u+1). \end{aligned}$$

The cost of this approach is equal to 3 multiplications in $\mathbb{F}_{2^{2\cdot 1223}}$ plus 9 additions in $\mathbb{F}_{2^{1223}}$. Using the cost of one multiplication in $\mathbb{F}_{2^{2\cdot 1223}}$ previously evaluated, we obtain a cost of 9 multiplications and 21 additions in $\mathbb{F}_{2^{1223}}$.

Cost of $f \cdot g$ *in Algorithm 1:* The most costly operation in the Miller's loop of Algorithm 1 is the multiplication $f \cdot g$. Thanks to the special form of $g = g_0 + g_1 u + v$, this multiplication can be reduced to two multiplications in $\mathbb{F}_{2^{2\cdot 1223}}$ plus a few additions. Indeed, if we write $f = f_0 + f_1 u + f_2 v + f_3 uv$, we have the following:

$$\begin{aligned} fg &= (f_0 + f_1 u)(g_0 + g_1 u) + (f_2 + f_3 u)(g_0 + g_1 u)v + (f_0 + f_1 u + f_2 v + f_3 uv)v \\ &= ((f_0 + f_1 u)(g_0 + g_1 u) + f_2 + f_3 + f_3 u) \\ &\quad + ((f_2 + f_3 u)(g_0 + g_1 u) + f_0 + f_3 + (f_1 + f_3)u)\, v \end{aligned}$$

This expression requires two multiplications, namely $(f_0 + f_1 u)(g_0 + g_1 u)$ and $(f_2 + f_3 u)(g_0 + g_1 u)$ in $\mathbb{F}_{2^{2\cdot 1223}}$, plus the additions of the resulting terms to $f_2 + f_3 + f_3 u$ and $f_0 + f_3 + (f_1 + f_3)u$. Consequently, the cost of $f \cdot g$ in Algorithm 1 is 6 multiplications and 15 additions in $\mathbb{F}_{2^{1223}}$.

4 Final Exponentiation

In this section, we focus on the final operation of the η_T pairing (Algorithm 1). This operation is to compute $(f^{(2^{2\cdot 1223}-1)(2^{1223}-2^{612}+1)})$ for a given $f \in \mathbb{F}_{2^{4\cdot 1223}}$.

We begin with the strategy presented in [12]: we first raise f to the power $2^{2\cdot1223}-1$ and then raise $f' = f^{(2^{2\cdot1223}-1)}$ to the power $(2^{1223}-2^{612}+1)$. This method is restated in Algorithm 2.

Algorithm 2. Final exponentiation

```
Require: $f \in \mathbb{F}_{2^{4\cdot1223}}$
Ensure: $f^{(2^{2\cdot1223}-1)(2^{1223}-2^{612}+1)}$
  Step 1. $S \leftarrow f^{2^{2\cdot1223}}$
  Step 2. $T \leftarrow f^{-1}$
  Step 3. $S \leftarrow S \times T$        // $S$ is equal to $f^{(2^{2^{2\cdot1223}}-1)}$
  Step 4. $T \leftarrow S^{2^{2\cdot1223}}$       // $T$ is equal to $(f^{2^{2^{2\cdot1223}}-1})^{2^{2\cdot1223}} = f^{1-2^{2^{2\cdot1223}}}$
  Step 5. $T \leftarrow T^{2^{612}}$       // $T$ is then equal to $f^{(2^{2^{2\cdot1223}}-1)(-2^{612})}$
  Step 6. $R \leftarrow S^{2^{1223}}$       // $R$ is equal to $f^{(2^{2^{2\cdot1223}}-1)(2^{1223})}$
  Step 7. $R \leftarrow R \cdot T \cdot S$      // $R$ is finally equal to $f^{(2^{2^{2\cdot1223}}-1)(2^{1223}-2^{612}+1)}$
  Step 8. return(R)
```

Note that in Step 4, we have used the fact that $f^{2^{4\cdot1223}} = f$, since $f \in \mathbb{F}_{2^{4\cdot1223}}$, to derive the expression $f^{1-2^{2^{2\cdot1223}}}$ of T. In Algorithm 2 a number of operation are performed including

– Powering to some 2 power exponents, like the power 2^{612} in Step 5 and the powers 2^{1223} and $2^{2\cdot1223}$ in Step 1, Step 4 and Step 6.
– Multiplication in the field $\mathbb{F}_{2^{4\cdot1223}}$ in Step 3 and Step 7.
– Inversion in $\mathbb{F}_{2^{4\cdot1223}}$ in Step 2.

We now specify how we perform the above operations. A multiplication in $\mathbb{F}_{2^{4\cdot1223}}$ is performed using the method given in Subsection 3.3. For the powering to 2^{1223} and $2^{2\cdot1223}$, we use the formula stated in the following lemma.

Lemma 1. *Let* $f = f_0+f_1u+f_2v+f_3uv \in \mathbb{F}_{4\cdot1223}$ *, then the following identities hold*

(i) $u^4 = u$ *and* $v^{16} = v$.
(ii) $f^{2^{1223}} = (f_0 + f_1 + f_2) + (f_1 + f_2 + f_3)u + (f_2 + f_3)v + f_3uv.$
(iii) $f^{2^{2\cdot1223}} = (f_0 + f_2) + (f_1 + f_3)u + f_2v + f_3uv.$

The proof is not difficult: *i*) is the direct consequence of the definition of u and v, and *ii*) and *iii*) are consequences of the little Fermat theorem and of *i*). Due to lack of space we omit the proof of this lemma.

For the other operations, i.e., inversion in $\mathbb{F}_{2^{4\cdot1223}}$ and exponentiation to the power 2^{612}, we propose some new optimizations. These optimizations are given in the following two subsections.

4.1 Inversion in $\mathbb{F}_{2^{4\cdot 1223}}$

To perform the inversion in $\mathbb{F}_{2^{4\cdot 1223}}$ we use an approach similar to the one used for some implementations of AES Sbox [5]. The proposed inversion is based on the following well known expression $A^{-1} = A^{r-1} \times (A^r)^{-1}$, which can be used to compute an inverse in an extension of degree two $\mathbb{F}_{q^2}$ over $\mathbb{F}_q$ by taking $r = q$. This reduces the computation of an inversion in $\mathbb{F}_{q^2}$ to an inversion in $\mathbb{F}_q$ as follows

$$A^{-1} = A^q (A^{1+q})^{-1},$$

since A^{1+q} is in $\mathbb{F}_q$. As A^{1+q} is a *norm* relative to the field extension $\mathbb{F}_{q^2}$ over $\mathbb{F}_q$, this approach is often referred to as the *norm approach* for field inversion. In our situation we apply this approach twice: first to reduce the inversion in $\mathbb{F}_{2^{4\cdot 1223}}$ to an inversion in $\mathbb{F}_{2^{2\cdot 1223}}$ with $q = 2^{2\cdot 1223}$ and then to reduce the latter inversion in $\mathbb{F}_{2^{2\cdot 1223}}$ to an inversion in $\mathbb{F}_{2^{1223}}$ with $q' = 2^{1223}$. This method is detailed in Algorithm 3.

Algorithm 3. Inversion in $\mathbb{F}_{2^{4\cdot 1223}}$

```
Require: $A = A_0 + A_1 u + A_2 v + A_3 uv$
Ensure: $A^{-1}$
    Step 1. $R_0 + R_1 u + R_2 v + R_3 uv \leftarrow (A_0 + A_2) + (A_1 + A_3)u + A_2 v + A_3 uv$    // $R = A^{2^{2\cdot 1223}}$
    Step 2. $S_0 + uS_1 \leftarrow (A_1 + uA_u)(R_1 + uR_u) + u(R_2 + uR_3)^2$    // $S = A \times A^{2^{2\cdot 1223}}$
    Step 3. $T_0 + T_1 u \leftarrow S_0 + S_1 + S_1 u$    // $T = S^{2^{1223}}$
    Step 4. $U \leftarrow S_0 T_0 + S_1 T_1$    // $U = S \times S^{2^{1223}}$
    Step 5. $V \leftarrow Inv_{\mathbb{F}_{2^{1223}}}(U)$    // $V$ the inverse of $U$
    Step 6. $W_0 \leftarrow T_0 V + T_1 V u$    // $W = T \times V$
    Step 7. $(Z_0 + Z_1 u) \leftarrow (R_0 + uR_1)(W_0 + W_1 u)$, $(Z_2 + uZ_{3)}) \leftarrow (R_2 + R_3 u)(W_0 + W_1 u)$  // $Z = R \times W$
    return($Z = Z_0 + Z_1 u + Z_2 v + Z_3 uv$)
```

We now evaluate the cost of Algorithm 3 in terms of the operations in $\mathbb{F}_{2^{1223}}$. Specifically, we count the number of additions (Add), squarings (Squ), multiplications (Mul) and inversion (Inv) in $\mathbb{F}_{2^{1223}}$. It is easy to see that Step 1 requires $2Add$ and Step 3 only $1Add$. Step 4 requires $1Mul$, $1Squ$ plus one Add (we have a squaring since $S_1 = T_1$), Step 5 requires one Inv and Step 6 requires $2Mul$. Finally, Step 2 and Step 7 contribute three multiplications in $\mathbb{F}_{2^{2\cdot 1223}}$, which leads to $9Mul + 9Add$ in $\mathbb{F}_{2^{1223}}$. The terms $u(R_2 + uR_3)^2 = R_2^2 u + R_3^2 \bmod (u^2 + u + 1)$ in Step 2 contributes to $2Squ + 2Add$. We finally obtain the overall cost of Algorithm 3: $14Add + 3Squ + 10Mul + 1Inv$.

In our proposed architecture, the additions and the squaring operations are performed through dedicated fully parallel adder and squarer. The multiplications are done through one of the $\mathbb{F}_{2^{1223}}$ multipliers presented in Section 3. We perform the inversion in $\mathbb{F}_{2^{1223}}$ using the algorithm of Itoh-Tsujii [14]. This method expresses the inversion as a sequence of squarings and multiplications specified by an addition chain. There exist several addition chains suitable to

Table 3. Complexity of the proposed approach for the final exponentiation

	#Add	*#Squ/SquRoot*	*#Mult*
Step 1.	2	0	0
Step 2.	14	1,225	24
Step 3.	21	0	9
Step 4.	2	0	0
Step 5.	2	2,444	0
Step 6.	6	0	0
Step 7.	42	0	18
Total	89	3669	51

perform an inversion efficiently. The addition chain we use to compute the inverse of a is as follows

$$\begin{aligned}&s \leftarrow a, s \leftarrow s^2 \cdot s,\ r \leftarrow s,\ s \leftarrow s^{(2^2)} \cdot s, s \leftarrow s^{(2^2)}, r \leftarrow r \cdot s, s \leftarrow s^{(2^4)} \cdot s,\\ &s \leftarrow s^{(2^8)} \cdot s, s \leftarrow s^{(2^{16})} \cdot s, s \leftarrow s^{(2^{32})} \cdot s, s \leftarrow s^{(2^4)}, r \leftarrow r \cdot s, s \leftarrow s^{(2^{64})} \cdot s,\\ &s \leftarrow s^{(2^{64})}, r \leftarrow r \cdot s, s \leftarrow s^{(2^{128})} \cdot s, s \leftarrow s^{(2^{256})} \cdot s, s \leftarrow s^{(2^{512})} \cdot s, s \leftarrow s^{(2^{128})},\\ &r \leftarrow r \cdot s, r \leftarrow r^2,\end{aligned}$$

and the last r satisfies $r = a^{-1}$. The resulting complexity of the inversion is equal to $14Mul$ and $1222Squ$ in $\mathbb{F}_{2^{1223}}$.

This means that the total cost in terms of additions, squarings and multiplications of the proposed inversion in $\mathbb{F}_{2^{4\cdot 1223}}$ is

$$14Add + 1225Squ + 24Mul.$$

In [12], the same inversion operation is performed by solving a system of equations and it incurs a cost of $57Add + 1230Squ + 50Mul$.

4.2 Complexity Evaluation and Comparison of the Final Exponentiation

Using the following expression given [2] of the exponentiation to the power 2^{612} in $\mathbb{F}_{2^{4\cdot 1223}}$ $(f_0+f_1u+f_2v+f_3uv)^{2^{612}} = (f_0^{2^{-611}})+(f_1^{2^{-611}})u+(f_2^{2^{-611}})+(f_3^{2^{-611}})uv.$ this exponentiation is reduced to four independent sequences of 611 square roots, and these four sequences can be performed in parallel. Based on this and the cost of inversion in $\mathbb{F}_{2^{4\cdot 1223}}$ from the previous subsection, in Table 3 we list the cost of each step of Algorithm 2 using the complexity results stated in Subsection 3.3, Lemma 1 and Subsection 4.1. The cost of each step is then added to obtain the overall cost of the proposed approach for the final exponentiation. We now compare the proposed approach with that of [12]. To this end, first we correct a small error in [12] which reports the 612 squarings for Step 5 as squarings in $\mathbb{F}_{2^{1223}}$ instead of squarings in $\mathbb{F}_{2^{4\cdot 1223}}$. Since a squaring in $\mathbb{F}_{2^{4\cdot 1223}}$ has a cost of $4Squ + 4Add$ in $\mathbb{F}_{2^{1223}}$, the actual complexity of the method of [12] is $3083Add + 3872Squ + 98Mul$. We then remark that the proposed approach reduces the number of additions, squarings and multiplications compared to that of [12].

5 Proposed Cryptoprocessor for η_T Pairing and Implementation Results

The final architecture for the η_T pairing is depicted in Fig. 3. There are three main blocks in our pairing-based cryptoprocessor, namely binary arithmetic unit (BAU), input and squaring unit (ISU) and data handling unit (DHU). The different computations involved in Algorithm 1, i.e., the operations of the Miller's loop and the operations of the final exponentiation, are distributed in the three blocks BAU, ISU and DHU. Specifically, BAU has five 1223-bit registers, one binary field multiplier, one adder and two squaring units. Both squaring units are connected in series to compute two squarings (X^{2^2}) in a single clock cycle during the computation of an inversion in $\mathbb{F}_{2^{1223}}$. Any of the registers R_0, R_1, R_2, R_3 and R_X can be added through the field adder by setting $R+1$ signal to high. R_2 is set to one at the beginning of computation as required by Algorithm 1. Note that our multiplier operates in steady state when the Miller's loop is computed. Hence the multiplication costs are only six and nine clock cycles in three-way split and two-way split multiplier structures, respectively. The ISU block has five 1223-bit registers to store the intermediate values x_1, x_2, y_1, y_2 and T values during the Miller's loop computation (cf. Algorithm 1). In final exponentiation, the computation $T^{2^{612}} \in \mathbb{F}_{2^{4\cdot 1223}}$ is performed through four sequences of 611 square roots in the ISU block. When extended field multiplication is performed in Miller's loop and final exponentiation, intermediate values are stored in eight 1223-bit registers in DHU. Powering to the exponent 2^{1223} during the final exponentiation are also computed in DHU. Furthermore, DHU handles transferring data from BAU to ISU and vice versa. There are two variants of this architecture: the three-way variant which uses the three-way hybrid sequential/parallel multiplier and the two-way variant which uses the two-way hybrid sequential/parallel multiplier.

In Table 4, we present the FPGA synthesis results of the two variants of the cryptoprocessor. We have run several test vectors through both implementations and both have been verified for correctness. In our design special attention was given to handle input and outputs in a more manageable way. This is because we have four 1223-bit inputs and four 1223-bit outputs. Current FPGA devices cannot handle such large amount of input/output in parallel. As a result, our VHDL codes were written for FPGA devices to support 306-bit words for input and output, i.e., an 1223-bit input is accepted in four steps. The results presented here do not include input and output timings and are purely on the computational time required to perform a single pairing.

The frequencies do not change compared to the frequencies of the multiplier (Table 2). Indeed the the critical path of the cryptoprocessor is part of the multiplier in $\mathbb{F}_{2^{1223}}$. The other parts of the architecture does not vary in the two considered variants of the cryptoprocessor, i.e., two-way and three-way variants. A consequence is that the difference in LUTs between the two architectures is roughly the same as the difference in LUTs of the multiplier in $\mathbb{F}_{2^{1223}}$ (cf. Table 2). We remark also that $\cong 80\%$ of the computation time is consumed by the Miller's loop and $\cong 20\%$ is consumed by the final exponentiation.

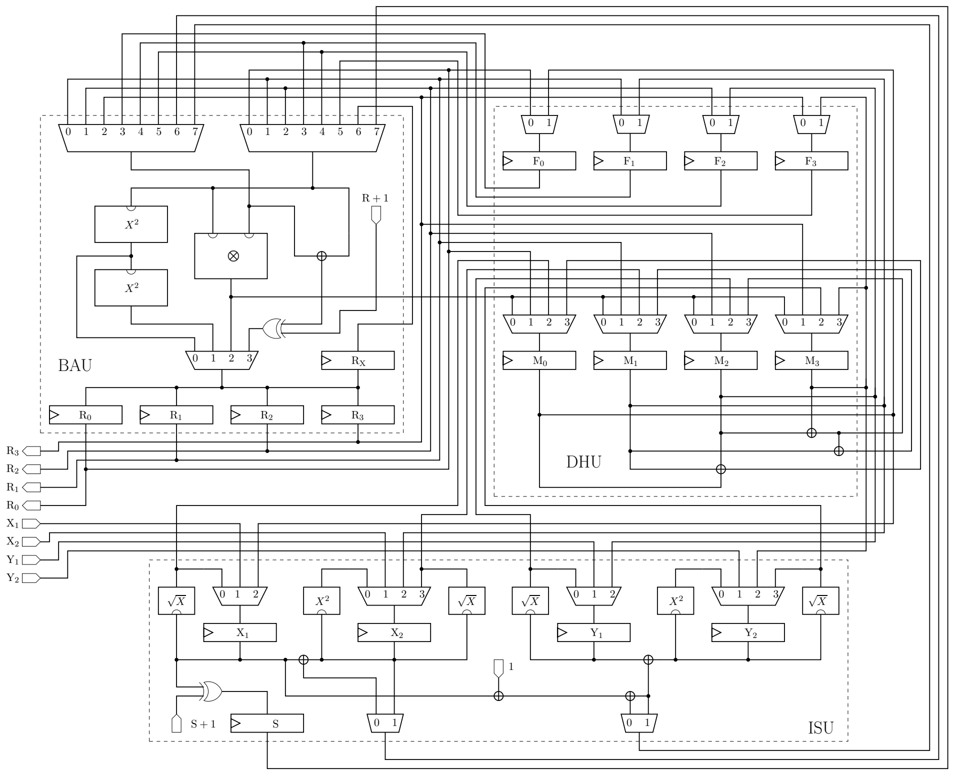


Fig. 3. Proposed η_T pairing architecture

Table 4. Implementation results for FPGA (Virtex-6)

Cryptoprocessor Design	Max. Freq. (MHz)	Processor (LUTs)	#CC per pairing	Multiplier (LUTs)	Time (μs)	#CC for all Mults.	#CC/ per Inverse	#CC per Ext. Mult.
Two-way way hybrid seq/para	271	50,179	40,320	19,721	148.78	38,562	794	89
Three-way hybrid seq/para	267	63,103	27,308	33,546	102.40	25,710	752	62

#CC denotes number of clock cycles.

In Table 5 we present our results for ASIC. The VHDL code is fed into Synopsys Design Compiler Version `E-2010.12` for synthesizing with TSMC 65 nm library `TCBN65GPLUS` at best case corner. At 500 MHz the architecture which uses the two-way sequential/parallel multiplier needs 80.64 μs and the architecture

Table 5. Implementation results for ASIC

Architecture	Gates	Area (μm^2)	Time (μs)	Power (mW)	Energy (μJ/*calc*)	*calc.s*/J
Two-way split	497,417	716,281	80.640	389.139	31.38	31,867
Three-way split	548,453	789,773	54.616	486.300	26.56	37,651

using the three-way sequential/parallel multiplier needs 54.616 μs per calculation. Energy consumption is calculated as *power* $\times$ *time* for a single computation and reported in Table 5. Obviously the architecture using the three-way sequential/parallel multiplier consumes more power, whereas a pairing based computation needs less energy in the same circuit, implying that the three-way approach is greener than the two-way.

6 Comparison and Conclusion

As the pairing algorithm used in the paper is heavily dominated by more than four thousand multiplications over $\mathbb{F}_{2^{1223}}$, it has thus been crucial to deploy high performance multiplier(s). To this end, the use of the Toeplitz matrix-vector product approach has enabled us to reduce the area and time requirements considerably. In order to explore area-time trade-offs for designs dealing with a large field like $\mathbb{F}_{2^{1223}}$, we have implemented both two-way and three-way split based multipliers in a hybrid sequential/parallel manner.

Using our two-way and three-way split multipliers, we have implemented the resulting η_T pairing cryptoprocessors in FPGA and ASIC. In Table 6 we have reported some known implementation results for pairing at the 128-bit security level. As it can be seen from Table 6, the previous best results is the one

Table 6. Comparison results for FPGA and ASIC

Design	Curve	FPGA	Area (Slices/DSP)	Freq.	Time (μs)	A (slice) $\cdot T$ (sec)
Fan *et al.* [11]	$E/\mathbb{F}_{p_{256}}$	xc6vx240-3	4,014/42	210	1,170	–
Ghosh *et al.* [13]	$E/\mathbb{F}_{p_{256}}$	xc4vlx200-12	5,2000	50	16,400	852.8
Estibals [7]	$E/\mathbb{F}_{3^{5\cdot 97}}$	xc4vlx200-11	4,755	192	2,227	10.6
Aranha *et al.* [1]	$Co/\mathbb{F}_{2^{367}}$	xc4vlx200-11	4,518	220	3,518	15.9
Ghosh *et al.* [12]	$E/\mathbb{F}_{2^{1223}}$	xc4vlx200-11	35,458	168	286	10.1
Ghosh *et al.* [12]	$E/\mathbb{F}_{2^{1223}}$	xc6vlx130t-3	15,167	250	190	2.9
This work/2-way	$E/\mathbb{F}_{2^{1223}}$	xc6vlx365t-3	13,596	271	148	2.0
This work/3-way	$E/\mathbb{F}_{2^{1223}}$	xc6vlx365t-3	16,403	267	102	1.7
Design	Curve	ASIC Tech.	Area (Gates)	Freq.	Time (μs)	A (gates) $\cdot T$ (sec)
Kammler *et al.*[15]	$E/\mathbb{F}_{p_{256}}$	130 nm CMOS	97,000	338	15,800	1,532.6
Fan*et al.* [10]	$E/\mathbb{F}_{p_{256}}$	130 nm CMOS	183,000	204	2,900	530.7
This work/2-way	$E/\mathbb{F}_{2^{1223}}$	65 nm CMOS	497,417	500	80.64	40.1
This work/3-way	$E/\mathbb{F}_{2^{1223}}$	65 nm CMOS	548,453	500	54.616	29.9

presented in [12] which requires 190 μs to complete a single computation of the η_T pairing. The work presented here requires 102 μs (~45% improvement) for the architecture using the three-way-sequential/parallel multiplier and 148 μs (~22% improvement) for the architecture using a two-way sequential/parallel multiplier. Our FPGA implementations also offer the best area-time products. Our ASIC implementations are faster and have better area-time products than those of the previous best implementations.

Acknowledgements. This work was supported in part by NSERC grants awarded to Dr. Hasan. The bulk of the work was done while Drs. Adikari and Negre were at the University of Waterloo.

References

1. Aranha, D.F., Beuchat, J.-L., Detrey, J., Estibals, N.: Optimal Eta Pairing on Supersingular Genus-2 Binary Hyperelliptic Curves. In: Dunkelman, O. (ed.) CT-RSA 2012. LNCS, vol. 7178, pp. 98–115. Springer, Heidelberg (2012)
2. Beuchat, J.-L., Detrey, J., Estibals, N., Okamoto, E., Rodríguez-Henríquez, F.: Fast Architectures for the η_T Pairing over Small-Characteristic Supersingular Elliptic Curves. IEEE Transactions on Computers 60(2), 266–281 (2011)
3. Boneh, D., Franklin, M.K.: Identity-Based Encryption from the Weil Pairing. SIAM Journal on Computing 32(3), 586–615 (2003)
4. Boneh, D., Lynn, B., Shacham, H.: Short Signatures from the Weil Pairing. Journal of Cryptology 17(4), 297–319 (2004)
5. Canright, D.: A very compact Rijndael S-box. Technical Report NPS-MA-04-001, Naval Postgraduate School (2004)
6. Cheung, R.C.C., Duquesne, S., Fan, J., Guillermin, N., Verbauwhede, I., Yao, G.X.: FPGA Implementation of Pairings Using Residue Number System and Lazy Reduction. In: Preneel, B., Takagi, T. (eds.) CHES 2011. LNCS, vol. 6917, pp. 421–441. Springer, Heidelberg (2011)
7. Estibals, N.: Compact Hardware for Computing the Tate Pairing over 128-Bit-Security Supersingular Curves. In: Joye, M., Miyaji, A., Otsuka, A. (eds.) Pairing 2010. LNCS, vol. 6487, pp. 397–416. Springer, Heidelberg (2010)
8. Fan, H., Hasan, M.A.: A New Approach to Sub-quadratic Space Complexity Parallel Multipliers for Extended Binary Fields. IEEE Transactions on Computers 56(2), 224–233 (2007)
9. Fan, H., Sun, J., Gu, M., Lam, K.-Y.: Overlap-free Karatsuba-Ofman polynomial multiplication algorithms. Information Security, IET 4, 8–14 (2010)
10. Fan, J., Vercauteren, F., Verbauwhede, I.: Faster $\mathbb{F}_p$-Arithmetic for Cryptographic Pairings on Barreto-Naehrig Curves. In: Clavier, C., Gaj, K. (eds.) CHES 2009. LNCS, vol. 5747, pp. 240–253. Springer, Heidelberg (2009)
11. Fan, J., Vercauteren, F., Verbauwhede, I.: Efficient Hardware Implementation of $\mathbb{F}_p$-Arithmetic for Pairing-Friendly Curves. IEEE Transactions on Computers 61(5), 676–685 (2012)
12. Ghosh, S., Roychowdhury, D., Das, A.: High Speed Cryptoprocessor for η_T Pairing on 128-bit Secure Supersingular Elliptic Curves over Characteristic Two Fields. In: Preneel, B., Takagi, T. (eds.) CHES 2011. LNCS, vol. 6917, pp. 442–458. Springer, Heidelberg (2011)

13. Ghosh, S., Mukhopadhyay, D., Roychowdhury, D.: High Speed Flexible Pairing Cryptoprocessor on FPGA Platform. In: Joye, M., Miyaji, A., Otsuka, A. (eds.) Pairing 2010. LNCS, vol. 6487, pp. 450–466. Springer, Heidelberg (2010)
14. Itoh, T., Tsujii, S.: A Fast Algorithm for Computing Multiplicative Inverses in GF(2^m) Using Normal Bases. Inf. Comput. 78(3), 171–177 (1988)
15. Kammler, D., Zhang, D., Schwabe, P., Scharwaechter, H., Langenberg, M., Auras, D., Ascheid, G., Mathar, R.: Designing an ASIP for Cryptographic Pairings over Barreto-Naehrig Curves. In: Clavier, C., Gaj, K. (eds.) CHES 2009. LNCS, vol. 5747, pp. 254–271. Springer, Heidelberg (2009)
16. Mastrovito, E.D.: VLSI Architectures for Computation in Galois Fields. PhD thesis, Linkoping University, Department of Electrical Engineering, Linkoping, Sweden (1991)
17. Barreto, P.S.L.M., Galbraith, S.D., O'Eigeartaigh, C., Scott, M.: Efficient pairing computation on supersingular Abelian varieties. Designs, Codes and Cryptography 42(3), 239–271 (2007)
18. Sunar, B.: A Generalized Method for Constructing Subquadratic Complexity GF(2^k) Multipliers. IEEE Transactions on Computers 53, 1097–1105 (2004)
19. Winograd, S.: Arithmetic Complexity of Computations. Society For Industrial & Applied Mathematics, U.S. (1980)

Feasibility and Practicability of Standardized Cryptography on 4-bit Micro Controllers

Nisha Jacob[1], Sirote Saetang[1], Chien-Ning Chen[2], Sebastian Kutzner[2], San Ling[1], and Axel Poschmann[1,2]

[1] School of Physical and Mathematical Sciences
Nanyang Technological University, Singapore
{njacob,sirote.tang,lingsan,aposchmann}@ntu.edu.sg
[2] Physical Analysis & Cryptographic Engineering (PACE)
Nanyang Technological University, Singapore
{chienning,skutzner}@ntu.edu.sg

Abstract. Myriads of ultra-constrained 4-bit micro controllers (MCUs) are deployed in (mostly) legacy devices, some in security sensitive applications, such as remote access and control systems or all sort of sensors. Yet the feasibility and practicability of standardized cryptography on 4-bit MCUs has been mostly neglected. In this work we close this gap and provide, to the best of our knowledge, the first implementations of `ECC` and `SHA-1`, and the fastest implementation of `AES` on a 4-bit MCU. Though it is not the main focus of this paper, we have investigated the SCA resistance trade-offs for `ECC` by implementing a variety of countermeasures. We hope that our comprehensive, highly energy-efficient crypto library—that even outperforms all previously published implementations on low-power 8-bit MCUs—will give rise to a variety of security functionalities, previously thought to be too demanding for these ultra-constrained devices.

Keywords: 4-bit MCU, AES, Elliptic Curve Cryptography, SHA-1, Lightweight Cryptography.

1 Introduction

Current market figures for embedded processors are hard to obtain in the open literature, nevertheless a book from 2003 indicates that 98 percent of all processors are embedded [48] and a publication from 1997 states with regards to the market shares of 4-, 8- and 16-bit micro controllers (MCUs): "*Like any ecosystem, the smallest creatures appear in the greatest numbers*" [47]. Though these market shares have certainly changed over the last 15 years—away from 4-bit MCUs and towards 8-, 16-, and 32-bit MCUs—it also indicates that myriads of 4-bit MCUs are deployed in legacy devices, e.g. watches and toys, but also security sensitive applications such as remote access and control systems, car immobilizers [45], one-time password generators, and all sorts of sensors.

8-bit MCUs have been long used as the platform of choice to evaluate the efficiency of cryptographic algorithms in embedded devices. On one hand this is a perfectly sound forward-looking approach, while on the other hand, this approach excludes myriads of 4-bit legacy devices, many of which could benefit from security functionalities. It is these legacy devices which make us believe that it is worth to evaluate the feasibility and practicability of cryptographic algorithms on 4-bit MCUs.

The feasibility of block cipher implementations on 4-bit MCUs has been shown in [50] using PRESENT [4] as an example. Shortly after, HUMMINGBIRD [13] — already broken [39] and replaced by a successor called HUMMINGBIRD-2 [14]— was implemented on the same 4-bit MCU [15]. Recently, also AES [34] as the first standardized algorithm[1] and PRINTcipher [24] have been implemented on the same platform [23].

In order to enable widespread security functionalities—such as legally binding sensor readings or secure firmware updates—not only block ciphers but also cryptographic hash functions and public key cryptography are required. To the best of our knowledge, no implementation on a 4-bit MCU has been published for the latter two categories so far. Hence, we close this gap by providing the first implementations of the SHA-1 hash function [35] and Elliptic Curve Cryptography (ECC) using the 160-bit curve secp160r1 [7]. We have also implemented AES, yielding the fastest implementation on a 4-bit MCU. All our highly-optimized implementations even outperform implementations on low-power 8-bit MCUs.

As side-channel attacks (SCA), such as Timing attacks [27], Simple Power Analysis (SPA) and Differential Power Analysis (DPA) [28], are a major concern for embedded devices, we have implemented basic countermeasures against them, though it is beyond the scope of this paper. As a matter of fact, our implementations of AES and SHA-1 are timing attack resistant, and for ECC we have investigated different DPA resistance-security trade-offs.

The remainder of this work is organized as follows. In Section 2 the target platform is briefly introduced before various implementations with different optimization goals of the AES and SHA-1 are presented in Sections 3.1 and 4, respectively. Subsequently, in Section 5 elliptic curve cryptography is treated. A wide variety of optimization tricks are described and different countermeasures against timing attacks, SPA and DPA are implemented. Then, in Section 6 our results are—where available—compared with previous implementations on 4-bit MCUs, and for the sake of completeness with implementations on 8-bit MCUs. Finally, Section 7 concludes this paper.

2 Target Platform and Design Flow

The Epson S1C63 family of MCUs was introduced in 2011 and is one of the most recent 4-bit low-power architectures. All members of the S1C63 family have a 4-bit core along with ROM, RAM, LCD drivers, I/O ports and a wide instruction set (mostly 1-2 cycles) with a linear addressing space without pages. It also has a two-stage pipeline (fetch and execute) and a maximum of 15 and 63 hardware

[1] PRESENT has been recently standardized as well [20].

and software interrupt vectors respectively, depending on the MCU being used. The MCUs differ mainly in the memory size ranging from 6kB to 50kB and on-board components, such as UART or hardware multiplier [41]. Our goal was to use the most constrained S1C63003 which has just 6kB of code ROM, 128 bytes of RAM and no data ROM. However, since this is not sufficient for the extensive computations of `ECC`, we moved to S1C63016 (only for ECC), which has 26kB of code ROM, 1kB of RAM and 2kB of data ROM. S1C63016 also has an integer multiplier/divider which is absent in S1C63003.

There are 47 basic types of instructions with 8 addressing modes (411 instructions in all). It has two 4-bit data registers `A` and `B`; two 16-bit index registers `X`, `Y` for 4-bit indirect data access; two stack pointers `SP1` (16-bit) and `SP2` (8-bit); a 4-bit condition flag register `F` which consists of extension `E`, interrupt `I`, carry `C` and zero flag `Z`; an 8-bit extend register `EXT` for extended addressing mode. Table 1 gives a list of the most frequently used instruction and its instruction cycles. Each instruction cycle is equal to 2 clock cycles[2].

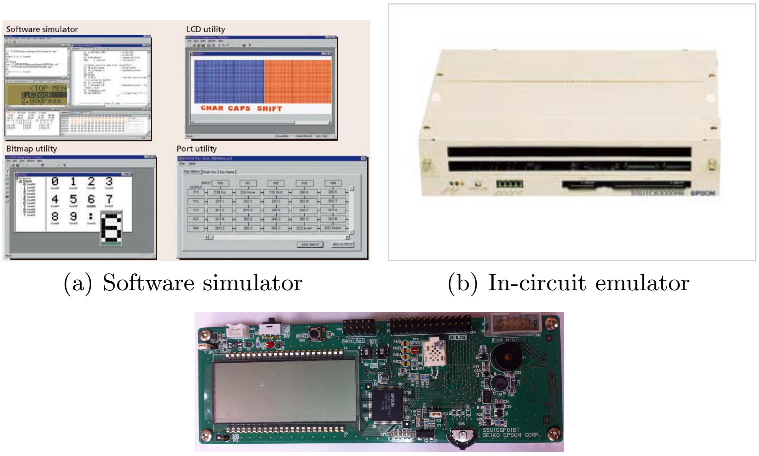
Table 1. Frequently used instructions [41] of EPSON S1C63 family MCU

Mnemonic*	Cycles
LD [%ir]+,%r; LD %r,[%ir]+; ADC %r,[%ir]; ADC %r, [%ir]+	1
AND %F,imm4; OR %F,imm4; CMP %r,%[ir]+; CMP [%ir]+,%r	1
JR sign8; JRNC sign8; JRNZ sign8; CALR imm8; RET	1
LDB % EXT, %BA; LDB %rr, imm8; ADD %ir, %BA; ADD %ir,sign8	1
LD [%ir]+,[%ir]+; LDB [%X]+,%BA; ADC [%ir]+,%r;	2
INC [addr6]; DEC [addr6]; XOR [%ir]+,%r; EX %r,[%ir]+	2
RETD	3

*ir = index register (X or Y); r= data register (A, B or F); addr6= 6-bit absolute data address; imm8= 8-bit immediate data; sign8= signed 8-bit digit; rr= XL,XH,YL,YH.

Details of the design flow of this MCU can be found in [44]. For debugging we use a software simulator on the PC that can even simulate an LCD (Fig.1(a)), and a hardware tool called *In-Circuit Emulator (ICE)*, an FPGA-based emulation board (Fig.1(b)) [43]. The code is first tested on the software simulator or on the ICE and can then be burned on the target board. The advantage of using the ICE over software simulator is to ensure the proper operation of the system before burning it onto the target board (Figure 1(c)). The software simulator and the debugger are used to get the cycle count and code size of the tested functions which are the two most common performance metrics for embedded platforms. Furthermore, we provide estimated energy consumptions based on public datasheets as an additional performance metric.

[2] In the remainder of the paper we refer to instruction cycles as cycles.



(a) Software simulator (b) In-circuit emulator

(c) Target board

Fig. 1. S1C63 family development tools [43]

3 The Advanced Encryption Standard

In this section we will first give a brief introduction to the Advanced Encryption Standard (AES). Subsequently, starting from a naïve implementation, we will describe several tricks applied for either code size or speed optimization. Table 2 summarizes our AES implementation results.

3.1 Introduction to AES

The Advanced Encryption Standard [34] is a symmetric block cipher with a block size of 128 bits and key sizes of 128, 192, and 256 bits, called AES-128, AES-192, and AES-256, respectively. The input and round keys are organized as a 4×4 state array, with each element being one byte. Encryption and decryption require $N_r = 10$, 12, and 14 rounds for AES-128, AES-192, and AES-256, respectively. Each round of AES is composed of the following four operations: *SubBytes*, *ShiftRows*, *MixColumns* and *AddRoundKey*. For details please refer to [11,34].

3.2 Naïve Implementation of AES

To implement AES on a 4-bit architecture, every byte of the state array and the key state array, $state_{rc}$ and key_{rc}, should be split into two 4-bit values. We use $state_{rcn}$ and key_{rcn}, where $n = 0$ or 1 denotes the high or low nibble. We now give a brief overview of our naïve AES implementation on Epson's 4-bit MCU.

SubBytes: In a straight forward implementation, each byte of the state array is substituted by using a look-up table, called S-Box, that consists of 256 8-bit entries. We split the S-Box into 16 tables, SBOX_i with $0 \leq i \leq 15$, each containing

16 entries. According to the manual of our target platform [41], table look-ups are realized by a combination of "`LD`", "`JR %BA`" and "`RETD`" instructions. It takes 11 cycles to substitute one byte, and in total *SubBytes* takes 163 cycles, i.e, 11 cycles for the first look-up, 10×15 cycles for the remaining look-ups (only 10 cycles/look-up because of the post-increment functionality), and 2 cycles for the call/return instruction and has a code size of 475 bytes.

ShiftRows: We exchange the location of elements in rows 1 to 3 of the state array nibble by nibble. For example, when exchanging $state_{100,101}$ and $state_{130,131}$, we have to copy $state_{100,101}$ to registers `A` and `B`, then copy $state_{130,131}$ to $state_{100,101}$. Finally, the $state_{130,131}$ is replaced by registers `A` and `B`. In total 90 cycles and 36 bytes of code size are required for *ShiftRows* and *InvShiftRows*.

MixColumns: This step is a matrix multiplication, using coefficients {01}, {02}, and {03} for encryption ({09}, {0B}, {0D} and {0E} for decryption). The element multiplication can be implemented by using only shifts and `XOR`s. In case the higher nibble of the multiplication result is greater than 7, we need a modulo reduction, that is an `XOR` with the constant "0x1B". To achieve a constant time implementation, we use a dummy reduction (`XOR` with "0x00") in case the higher nibble is less or equal than 7. In our constant time implementation, in total *MixColumns* requires 1297 cycles and 555 bytes of code size, and *invMixColumns* requires 6207 cycles and 1554 bytes of code size.

AddRoundKey: It performs a nibble-wise `XOR` between the state array and key state arrays. In total 113 cycles and 43 bytes of code size are required for the *AddRoundKey* operation. After every *AddRoundKey* operation, the *Key Schedule* is performed to generate the next round key. In total, *Key Schedule* requires 204 cycles and 114 bytes of code size for encryption (276 cycles and 150 bytes of code size for decryption).

3.3 Memory-Optimized Implementation of AES

In order to reduce the memory requirement, we have applied the following optimization tricks.

Code Structure Optimization Process: We need the extended register for call and jump instructions if the relative address is more than 8 bits. These extended instructions consist of two "regular" instructions, thus doubling the code size. Hence, we rearranged parts of the code in memory to avoid these extended instructions. Furthermore, we combined *MixColumns*, *Key Expansion*, and *AddRoundKey* operations in the same function for AES-128 and AES-256, but not for AES-192 due to the different Key Expansion routine. We also combined *SubBytes* and *ShiftRows* into one function to save code space.

MixColumns/InvMixColumns Optimization Process: We have implemented the technique proposed in [11] but did not achieve significant code savings for encryption. Hence, we stuck to the direct multiplication. For decryption, this technique will reduce the code size by 30%, and it is 3.5 times faster than the direct multiplication.

Key Expansion Optimization Process: In our naïve implementation, every byte of the last column of the key state array is loaded from RAM to process substitution, rotation, and `XOR` with the round constants and then stored back to the key state array. The "`EX`" instruction is used to exchange values between RAM and the registers `A` and `B`. When using this "`EX`" instruction efficiently one can combine a *load from RAM* and *store to RAM* into one instruction, hence halving the number of instructions required for loading and saving the state.

3.4 Speed-Optimized Implementation of `AES`

In order to speed up the naïve implementation as much as possible, we have applied the following optimization tricks.

Code Structure Optimization Process: To speed up our implementation we used two well-known techniques called loop-unrolling and inline expansion. When unwinding a loop, the instructions are replicated as many times as the loop will be processed, to avoid the overhead due to conditional branching. Inline expansion is a similar technique: it replaces the function call by the actual contents of the to-be-called function, thus saving the function call.

MixColumns/InvMixColumns Optimization Process: By using look-up tables for multiplication by {02}, {03}, {09}, {0B}, {0D} and {0E} we could speed up *MixColumns* by 29% and *invMixColumns* by a factor of more than 3. These gains come at an additional memory cost of 72% and 32%, respectively.

3.5 Implementation Results

All implementations of `AES` are tailored for the most constrained member of the MCU family, the Epson S1C63003 MCU running at 3 Volt. Table 2 compares our naïve and optimized implementations. For encryption, we could reduce the code size up to 15% for the memory-optimized implementation. For the speed-optimized version, we could boost the speed of the algorithm by almost 22%. For decryption, we could reduce the code size by almost 15% for the memory-optimized version and could boost the speed by almost 72% for the speed-optimized one. In addition, we also calculated the throughput of all versions and the energy consumption for computing one block of `AES`. The power consumptions were estimated based on information from datasheets or from information directly received from the manufacturer by using the formula voltage * current * (cycles*2) / frequency.

We have also `AES` implementations providing both encryption and decryption functionality. The memory-optimized version requires 2625, 2999, and 2895 bytes (about 10% more than the respective decryption-only variant), and the speed-optimized variant requires 4674, 4962, 5698 bytes (about 12% more than the respective decryption-only variant).

4 `SHA-1`

In this section we will first give a brief introduction into the Secure Hash Algorithm (`SHA-1`). Subsequently, starting from a naïve implementation, we will describe several tricks that we applied to optimize the implementation regarding to code size and speed.

Table 2. Implementation results of AES

Encryption		AES-128			AES-192			AES-256		
		Naïve	Mem Opt	Spd Opt	Naïve	Mem Opt	Spd Opt	Naïve	Mem Opt	Spd Opt
Code size [bytes]		1,426	1,294	2,645	1,715	1,468	2,942	1,677	1,458	3,280
Cycles		17,675	17,347	13,729	21,188	20,883	16,524	24,979	24,771	19,474
T'put [kbps]	@32.768KHz	0.1159	0.1181	0.1492	0.0967	0.0981	0.1239	0.0820	0.0827	0.1052
	@4MHz	14.144	14.412	18.210	11.799	11.971	15.130	10.008	10.092	12.838
Energy [μJ/block]	@32.768KHz	7.44	7.31	5.78	8.92	8.79	6.96	10.52	10.43	8.20
	@4MHz	5.83	5.72	4.53	6.99	6.89	5.45	8.24	8.17	6.43
Decryption										
Code Size [bytes]		3,062	2,462	4,152	3,416	2,670	4,542	3,382	2,698	4,825
Cycles		64,636	23,901	18,688	78,315	28,552	22,120	92,378	33,554	26,242
T'put [kbps]	@32.768KHz	0.0317	0.0857	0.1096	0.0262	0.0717	0.0926	0.0222	0.0610	0.0780
	@4MHz	3.868	10.460	13.378	3.192	8.756	11.302	2.706	7.451	9.527
Energy [μJ/block]	@32.768KHz	27.22	10.07	7.87	32.98	12.02	9.32	38.90	14.13	11.05
	@4MHz	21.33	7.89	6.17	25.84	9.42	7.30	30.48	11.07	8.66

4.1 Introduction to SHA-1

SHA-1 is one of the Secure Hash Standards published by the National Institute of Standards and Technology (NIST) [35] in 1995. SHA-1 is an iterative, one-way hash function that can process messages up to a length $< 2^{64}$ bits and produces a 160-bit message digest. According to [35], there are two stages of SHA-1, i.e., *preprocessing* and *hash computation*.

Preprocessing stage: It consists of the following 3 steps.

1. *Padding*: The message is padded by a single '1' bit followed by the necessary number ($0 \sim 511$) of '0' bits, and then the bit length of the original message, represented by a 64-bit integer, is appended. After padding, the bit length will be a multiple of 512.
2. *Parsing the padded message*: This step divides the message into N blocks, each block containing 512 bits.
3. *Initialize Hash Value*: In this step the 160-bit starting value is initialized by five 32-bit words as follows: A = 0x67452301, B = 0xEFCDAB89, C = 0x98BADCFE, D = 0x10325476 and E = 0xC3D2E1F0.

Hash computation: This stage is the heart of SHA-1 and consists of 80 rounds. Each round (see Fig. 2) uses a function $f_t(B, C, D)$ and processes constants (K_t) as described in Table 3.

After processing all N blocks, the resulting 160-bit message digest is the concatenation of $A\|B\|C\|D\|E$. For further details, please refer to [35].

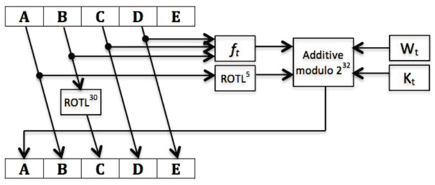


Fig. 2. One round of `SHA-1` computation

Table 3. `SHA-1` function ($f_t(B, C, D)$) and constants (K_t)

Round (t)	$f_t(B, C, D)$	K_t
0 to 19	$(B \wedge C) \oplus (\neg B \wedge D)$	`0x5A827999`
20 to 39	$B \oplus C \oplus D$	`0x6ED9EBA1`
40 to 59	$(B \wedge C) \oplus (B \wedge D) \oplus (C \wedge D)$	`0x8F1BBCDC`
60 to 79	$B \oplus C \oplus D$	`0xCA62C1D6`

4.2 Implementation of `SHA-1`

There are two different methods to compute message scheduling as mentioned in [35]. We choose the method for devices with limited memory.

There are five operations required for `SHA-1`: *AND, NOT, XOR, Left Rotation* and *Modulo Addition*. However, as the MCU does not support *NOT*, we use *XOR* with `0xF` instead. The 5-bit left rotation is implemented by first swapping 4-bit words within the 32-bit chunk and then left rotating the result by 1-bit. For the 30-bit left rotation, we apply a similar technique, i.e., first swapping 4-bit words within the 32-bit chuck to get a 28-bit left rotation and then doing a 1-bit left rotation twice. The 5-bit left rotation requires 223 bytes of code and 180 clock cycles. The 30-bit left rotation reuses some sub-functions in the 5-bit left rotation, so it requires additional 21 bytes and in total takes 301 cycles. In addition, the 32-bit modulo addition requires 82 bytes of code and 93 cycles.

Memory-Optimized Implementation: As for `AES` we tried to avoid extended jumps and calls by rearranging parts of the code. Furthermore, in the specification of `SHA-1` it can be seen that the round function of rounds 60 to 79 uses the same round function as rounds 20 to 39. Thus, we can use the same function twice and hence, save 13 bytes of program memory at no additional timing cost.

Speed-Optimized Implementation: We have applied 2-bit right rotation instead of 30-bit left rotation. As a consequence, we could speed up this part by 17%, while the code size increased a mere 7% as compared to the memory-optimized version. We also applied loop-unrolling where possible and inline expansion of the function *left rotation*, *right rotation* and *modulo addition*. Additionally, we also applied the trick for minimizing the round function as mentioned in [31] to boost up the speed.

4.3 Implementation Results

Our SHA-1 can be implemented on the smallest Epson S1C63003 MCU running at 3 volts. Detailed results are provided in Table 8 in Sec. 6. As can be seen the speed-optimized version requires 14% more memory, but it can boost the speed by almost 24% in comparison to the memory-optimized implementation.

5 Elliptic Curve Cryptography

Elliptic curve cryptography (ECC) [26,32] is a public-key cryptography with much smaller key sizes than RSA or other public-key systems based on the discrete logarithm problem. Because of its small key sizes, it is very suitable for constrained devices. ECC can be used to implement a wide variety of applications, such as Diffie-Hellman key exchange (e.g. ECDH [40]) or digital signatures (e.g. ECDSA [21]) amongst others.

ECC relies upon group operations in elliptic curve group, and a group $\mathbb{E}$ over field $\mathbb{F}_p$ can be defined by the points (x, y) satisfying the short Weierstrass form:

$$\mathbb{E} : y^2 = x^3 + ax + b, \text{ where } 4a^3 + 27b^2 \neq 0.$$

In “Standards for Efficient Cryptography 2” [7] a curve is specified by a sextuple: (p, a, b, G, n, h), where p is the prime, a and b are the curve parameters, G is a base point with order n, and h is the cofactor. In this paper, we have implemented ECDH using the 160-bit curve secp160r1 of SEC 2 on the S1C63016 MCU.

5.1 Efficient Implementation of ECC

The computation in ECC can be divided into three layers: prime field arithmetic at the bottom, point arithmetic in the middle, and protocol on top. The three layers are implemented separately and the implementations at the lowest level greatly impacts the overall run time, hence optimizations at this level are crucial.

Prime Field Arithmetic. The curves suggested by SEC 2 employ *pseudo Mersenne primes*, e.g., $p = 2^{160} - 2^{31} - 1$ in secp160r1. Modulo operations will be much more efficient than using the Barrett reduction [3] or the Montgomery multiplication [33]. We have implemented the following operations in this layer.

Modular addition/subtraction: The naïve addition/subtraction with reduction takes at most 428 cycles. It might be susceptible to timing attacks because of the unfixed run time. By using some extra data RAM a fixed run time of 292 cycles is achieved. Besides, the code size can be optionally reduced from 352 to 291 bytes along with the data RAM usage increasing from 40 to 60 bytes.

Multiplication (M) *and Squaring* (S): According to our implementation results (Table 4), row-wise (operand scanning form) multiplication is more suitable for the 4-bit architecture than column-wise (product scanning form) and hybrid [49] algorithms. When using row-wise multiplication, the first operand remains unchanged for the whole row, and only the second operand needs to be updated.

This is why, compared to the column-wise and hybrid method, the row-wise method requires less IO operations and hence resulting in a better performance.

The main bottlenecks on this 4-bit MCU are carry handling and the slow multiplication instruction[3]. The first one can be solved by using additional 160-bit of memory to accumulate the results of atomic multiplications in one round of the row-wise multiplication instead of adding them to the final result separately. The second one can be solved by interleaving multiplication with other instructions. These two improvements together deliver about twice the performance, i.e., reducing the run time from 28,680 to 16,690 cycles with a code size of 324 bytes and an additional 10-byte data RAM. A separate squaring algorithm [6] further improves the performance, each squaring takes 12,985 cycles at an additional cost of 330 bytes of code size.

Table 4. Implementation results of field arithmetic

Naïve implementation	No. of cycles	Code size [bytes]
Row-wise [19]	28,680	300
Column-wise[19]	30,953	344
Hybrid [49]	29,485	368

Bisection: For an even number, division by 2 is a right shift of its binary representation. For an odd number, an extra addition of the prime p is required before the right shift.

Inversion (I): We have implemented the binary extended GCD algorithm [25, Ch 4.5.2] which requires multi-precision addition/subtraction and right shift (division by 2).

Reduction: Because of the pseudo Mersenne prime $p = 2^{160} - 2^{31} - 1$ in the curve secp160r1, the 320-to-160-bit reduction can be evaluated by shift and add. A naïve implementation of reducing from MSB to LSB takes 1,604 cycles with a code size of 775 bytes, and most of them are spent on the carry handling. In order to avoid this bottleneck, we rearrange the reduction as

$$\begin{aligned} H \times 2^{160} + L &\equiv H \times (2^{31} + 1) + L \\ &\equiv H + L + (H \lll 31) + H_{159\sim 129} \times 2^{31} \pmod{p}, \end{aligned} \quad (1)$$

where H and L are the most and least significant 160 bits of the 320-bit value respectively, "$\lll 31$" is a 31-bit left rotation, and $H_{159\sim 129}$ is the most significant 31 bits of H. By accumulating the carries of the three additions and then handling them together, the performance will be significantly improved, and it requires only 679 cycles and 624 bytes.

Point Arithmetic. The major computation in ECC is the point multiplication nP which can be evaluated through the combination of point doubling $2P$ and

[3] Refer to the specifications in the Epson technical manual [42].

point addition $P_1 + P_2$. Instead of representing points in affine coordinates ($\mathcal{A}$), we employ Jacobian projective coordinates ($\mathcal{J}$) to implement point doubling and addition. A point (x, y) in $\mathcal{A}$ can be represented by $(x, y, 1)$ in $\mathcal{J}$, and a point (X, Y, Z) in $\mathcal{J}$ is identical to the point $(X/Z^2, Y/Z^3)$ in $\mathcal{A}$. The advantage of using projective coordinates is that we only need 1I after finishing the point multiplication instead of 1I for every point doubling and point addition in affine coordinates. The following are the three point operations, and detailed results are summarized in Table 5 and Table 6.

Point doubling (D): We implement point doubling in Jacobian coordinates ($2\mathcal{J} \rightarrow \mathcal{J}$). It requires 4M and 4S as well as some minor field operations, or alternatively 3M and 5S by using the trick $\alpha\beta = \frac{1}{2}\left((\alpha+\beta)^2 - \alpha^2 - \beta^2\right)$ [29]. Since in our implementation 1S $\approx$ 0.7M, this trick achieves 2% speed-up along with 3% increment in code size.

Point addition/subtraction (A): Point addition in mixed coordinates ($\mathcal{J} + \mathcal{A} \rightarrow \mathcal{J}$) takes 8M and 3S, or 7M and 4S by using the trick described above. Point subtraction is similar to point addition but has an extra subtraction to calculate the y-coordinate.

Point Multiplication (PM): The left-to-right binary PM requires 159D and on average 80A. When recoding the scalar to non-adjacent form (NAF) [36], PM requires 159D and on average 53A, that is, about 14% speed-up compared to the one with a binary scalar. However, the NAF algorithm recodes from right to left and needs more data RAM to separately store the NAF-recoded signed-digit scalar. To overcome this, we have employed the left-to-right analogue of NAF [22] to achieve PM along with on-the-fly scalar recoding.

Table 5. Implementation results of point arithmetic

Operation	Description	No. of cycles	Code size [bytes]
Point doubling	4M + 4S	128,453	1,038
$2\mathcal{J} \rightarrow \mathcal{J}$	3M + 5S	125,883	1,064
Point addition	8M + 3S	183,698	1,945
$\mathcal{J} + \mathcal{A} \rightarrow \mathcal{J}$	7M + 4S	181,343	1,994

Protocol Layer. Optimizations at this level depend on the protocol being implemented. We focus on the Elliptic Curve Diffie-Hellman key exchange (ECDH) [19]. Each key exchange requires 2PM and 2I.

5.2 Side Channel Resistant Implementations of ECC

The immunity against side-channel attacks is also investigated. All prime field arithmetic in our implementation takes a fixed amount of time and is resistant to timing attacks. We also investigate the performance and side-channel immunity of various PM algorithms in the curve arithmetic.

Double-and-add-always algorithm [10]: It achieves a regular computational sequences by introducing dummy operations. A regular computational sequence

Table 6. Implementation results of point multiplication and SCA

Point multiplication algorithm	Cycles [millions]	Code size [bytes]	Side-channel immunity
Left-to-right `PM`	35.20	7,201	-
Left-to-right `PM` with left-to-right NAF	30.39	10,387	-
Double-and-add-always	49.48	8,042	SPA
BRIP	50.00	8,080	SPA,RPA,ZPA,DPA
Randomization of scalar (20-bit)	33.21	10,404	DPA
Randomization of scalar (64-bit)	43.04	10,531	DPA
Randomized projective coordinates	31.50	10,282	DPA
Randomization of scalar (20-bit) & Randomized projective coordinates	33.52	10,501	DPA

can prevent SPA but it is still vulnerable to DPA and other sophisticated attacks, e.g., Refined Power Analysis (RPA) [17] and Zero Power Analysis (ZPA) [1].

Binary expansion with random initial point (BRIP) [30]: In BRIP, the intermediate result of `PM` is blinded by a random point R. It is also a double-and-add-always algorithm but without dummy operations and thus making it resistant to even DPA, RPA and ZPA.

Randomization of scalar [10]: The scalar d is randomized by $d' = d + n \times \#\mathbb{E}$, where $\#\mathbb{E}$ is the curve order and n is a random number. There is an inherent security-efficiency trade-off: Using, say, a 20-bit random number already provides a decent level of DPA resistance as it multiplies the effort of an attacker by 6 orders of magnitude while having a mere 10% overhead in execution time.[4] For a highly secure implementation using a 64-bit random number the timing increases by around 42%.

Randomized projective coordinates [10]: A point $P = (x, y)$ can be represented in Jacobian projective coordinates as (r^2x, r^3y, r) for any $r \in \mathbf{Z}_p^*$. When computing dP by the left-to-right scalar algorithm, the temporary result T is initialized by $T = P$, and in each iteration, updated by $T = 2T$ and optionally $T = T + P$. In our implementation, instead of randomizing P, we only randomize the initial value of T, i.e., P is still of affine coordinates, to avoid the much more expensive point addition in Jacobian coordinates ($\mathcal{J} + \mathcal{J} \to \mathcal{J}$, 12`M` and 4`S`).

Table 6 summarizes the implementation results of various point scalar multiplication algorithms. All of them are implemented by using Jacobian projective coordinates. In randomization of scalar and randomized projective coordinates, the scalar will be also recoded by the left-to-right NAF method. As one can see, BRIP is the most secure one but also the slowest. For constrained devices, we suggest using 20-bit randomization of scalar along with randomized projective coordinates, as it provides basic resistance against DPA and also increases the difficulty for SPA [8] due to the NAF-recoded scalar.

[4] Assuming a successful DPA would require 100 measurements, it would take around 54 years to *measure* the power consumption compared to 26 minutes without randomization.

6 Discussion

In this work we have already shown the feasibility of implementing symmetric and asymmetric cryptography on 4-bit MCUs. What is left to be shown is the practicability and meaning of cryptography on 4-bit MCUs in real-world scenarios as well as the performance in comparison to common 8-bit implementations regarding three important criteria for embedded systems: *energy consumption, speed and code size*. As already pointed out in Section 2, we use the software simulator and the debugger to obtain the cycle count and the code size. Power figures from the data sheet are used to estimate the energy consumption of our implementations.

6.1 AES

We compare our 4-bit AES implementation with the only other 4-bit AES implementation known so far [23] and with two recent 8-bit AES implementations [5,37]. The MCUs used in [5,37] belong to the megaAVR family, which are popular 8-bit processors used in research as well as in embedded devices. The most notable difference to our 4-bit MCU is that the megaAVRs have 32 8-bit registers which can all be used as destination registers of arithmetic operations. In addition, the megaAVRs support the pre-decrement and post-increment functionality, whereas our 4-bit MCU only supports the post-increment functionality.

Table 7 shows the results regarding speed, code size and energy consumption for different optimization strategies. It should be noted that the comparison is not completely fair since the ATmega128 and the AT90USB162 are not the lowest-power devices but still are stated as low-power MCUs by Atmel. Note further that it was not possible to compare the energy consumptions at the same frequency because the manufacturers only provide numbers for one (the most typical) frequency of their MCUs. To estimate the energy consumption of our 4-bit MCU we chose 4 MHz because first, at this frequency our implementations can be directly compared to most of the 8-bit implementations in the following sections, and second, this configuration is also the most energy efficient.

Table 7. Comparison of different AES implementations

	MCU	Opt.	Enc. [cycles]	Dec. [cycles]	Code size [bytes]	Energy** [μJ/block] @ 3V
4-bit	Epson S1C63	Speed	13,929	18,688	4,674	4.6 @ 4MHz
		Size	16603	23,901	2,625	5.5 @ 4MHz
	MARC4 [23]	Speed	15,977	20,801	3,679	14.3 @ 1MHz
		Size	24,012	34,432	1,775	21.6 @ 1MHz
8-bit	ATmega128 [37]	-	3,766	4,558	3,410	14.1 @ 4MHz
	AT90USB162 [5]	Speed	2,740	3,648	1,912	7.2 @ 8MHz
		Size	2,942	3,690	1,912*	7.7 @ 8MHz

* uses less RAM. ** energy consumption for encryption.

Not surprising, the 8-bit implementations of AES beat the 4-bit implementations by an order of magnitude in terms of speed since the AES operations are predestined to be very efficient on 8-bit architectures. But, as one can see, our implementation beats the implementation on the MARC4 in terms of speed. Hence, we can claim that this implementation is the fastest 4-bit implementation of AES up to now. More importantly, our implementation is the most efficient one in terms of energy consumption as it consumes nearly half the energy per block in comparison to the best 8-bit implementation and it outperforms the other implementations by a factor of approximately three. This can be seen as a strong argument for implementing cryptography on a 4-bit MCU since energy consumption is arguably one of the most important criteria for low-power devices.

6.2 SHA-1

We only found two references for SHA-1 implementations on an 8-bit MCU in [38], which are compared to our 4-bit implementation in Table 8.

Table 8. Comparison of different SHA-1 implementations

	MCU	Opt.	Cycles per block	Code size [bytes]	Energy [μJ/block] @ 3V
4-bit	Epson S1C63	Speed	87,788	2,324	29.0 @ 4 MHz
		Size	108,666	2,038	35.9 @ 4 MHz
8-bit	ATmega128 [16]	Generic	63,000	4,000	236 @ 4 MHz
	megaAVR family [12]		37,030	1,022	139 @ 4 MHz

*Since [12] is a generic implementation, we used the properties of the ATmega128 to estimate the energy consumption.

As can be seen our implementation takes roughly double the clock cycles and double the size compared to the best 8-bit implementation. This is due to the fact that SHA-1 mostly uses primitive instructions (1 cycle) on 4- or 8-bit of data. Hence, we need roughly twice the amount of instructions resulting in twice the amount of cycles and code size. But again, our implementation on the 4-bit MCU outperforms the other implementation in terms of energy consumption by a factor of almost five.

6.3 ECC

In this section we compare our 4-bit ECC implementation regarding speed and code size with three other 8-bit implementations using the same curve, i.e. secp160r1, found in [18] and [9]. [18] describes two implementations on two different MCUs, namely the Chipcon CC1010 and the Atmel ATmega128. [9] describes a ECC implementation on an Atmel ATmega8 [2] which is very similar to the ATmega128 as both belong to the megaAVR family.

The Chipcon CC1010 is an 8-bit MCU which implements the 8051 instruction set [46]. The most notable properties are that every instruction cycle takes 4 clock cycles, hence quartering the effective frequency. Second, the 8-bit accumulator is the destination register for all arithmetic functions resulting in the same complexity for storing or further processing intermediate results as on our 4-bit MCU, thus making the CC1010 a perfect candidate for the comparison.

As can be seen in Table 9 a field multiplication on the 4-bit MCU takes around 8 times as many instruction cycles as the fastest implementation on an 8-bit controller and twice as many cycles as on the CC1010. This is due to the fact that a multiplication instruction on the Epson S1C63 requires 10 cycles, compared to only 2 cycles and 5 cycles on the ATmega and on the CC1010, respectively. This fact also leads to a higher time share of the field multiplication.

As a consequence, our fastest point multiplication requires roughly 5 times more instruction cycles than the fastest ATmega implementation and twice as many instruction cycles as the CC1010. But, as for AES and SHA-1, the 4-bit implementation beats the 8-bit implementations by a factor of more than two regarding the energy consumption.

Table 9. Comparison of ECC implementations

	MCU	Point Mul. [cycles in M]	Field Mul. [cycles]	Time share in PM	Code size [bytes]	Energy [mJ/block] @ 3V
4-bit	Epson S1C63	30.39	16,690	88%	10,616	10.02 @ 4 MHz
8-bit	ATmega128 [18]	6.48	1,986	77%	3,682	24.3 @ 4 MHz
	ATmega8 [9]	10.4	n/a	n/a	7,080*	28.08 @ 4 MHz
	CC1010 [18]	16.88	9,321	85%	2,166	253.2 @ 4 MHz

* includes ECDSA.

7 Conclusion

In this work we have studied the feasibility and practicability of standardized cryptography on a 4-bit MCU. We have provided the fastest implementation of AES, and the first implementations of SHA-1 and ECC, this way proving their feasibility on an ultra-constrained 4-bit MCUs. More interesting, however, is that all three primitives require between 39% to 79% less energy than previously published implementations on low-power 8-bit MCUs. Our crypto modules can be combined to enable a wide range of different applications. For example, non-timing critical applications, such as legally binding sensor readings or secure firmware updates can be achieved using ECDSA [21] by combining our SHA-1 and ECC modules.

Though it is not the main focus of this paper, we have investigated the SCA resistance trade-offs for ECC by implementing a variety of countermeasures. Future work includes implementation of SCA countermeasures for AES and evaluation of our implemented SCA countermeasures for ECC and AES by conducting physical experiments, which was beyond the scope of this work. Another interesting topic might be the analysis of the resistance against fault attacks of 4-bit MCUs.

Acknowledgement. The authors were supported in part by the Singapore National Research Foundation under Research Grant NRF-CRP2-2007-03. The authors would also like to thank Andreas Enge for many fruitful discussions.

References

1. Akishita, T., Takagi, T.: Zero-Value Point Attacks on Elliptic Curve Cryptosystem. In: Boyd, C., Mao, W. (eds.) ISC 2003. LNCS, vol. 2851, pp. 218–233. Springer, Heidelberg (2003)
2. Atmel Corporation. ATmega8/ATmega8L datasheet (February 2011), http://www.atmel.com/Images/doc2486.pdf
3. Barrett, P.: Implementing the Rivest Shamir and Adleman Public Key Encryption Algorithm on a Standard Digital Signal Processor. In: Odlyzko, A.M. (ed.) CRYPTO 1986. LNCS, vol. 263, pp. 311–323. Springer, Heidelberg (1987)
4. Bogdanov, A.A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M., 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), http://lightweightcrypto.org/present/
5. Bos, J.W., Osvik, D.A., Stefan, D.: Fast implementations of AES on various platforms. Cryptology ePrint Archive, Report 2009/501 (2009)
6. Brown, M., Hankerson, D., López, J., Menezes, A.: Software Implementation of the NIST Elliptic Curves Over Prime Fields. In: Naccache, D. (ed.) CT-RSA 2001. LNCS, vol. 2020, pp. 250–265. Springer, Heidelberg (2001)
7. Certicom Research. Standards for efficient cryptography, SEC 2: Recommended elliptic curve domain parameters (2000)
8. Chen, C.-N., Yen, S.-M., Moon, S.-J.: On the computational sequence of scalar multiplication with left-to-right recoded NAF and sliding window technique. IEICE Transactions 93-A(10), 1806–1812 (2010)
9. Chmielowiec, A.: Elliptic curve cryptography in small devices, http://students.mimuw.edu.pl/~ac181080/data/ecc_in_small_devices.pdf
10. Coron, J.-S.: Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems. In: Koç, Ç.K., Paar, C. (eds.) CHES 1999. LNCS, vol. 1717, pp. 292–302. Springer, Heidelberg (1999)
11. Daemen, J., Rijmen, V.: The Design of Rijndael: AES - The Advanced Encryption Standard. Springer (2002)
12. Das Labor. Crypto-avr-lib (January 2008), http://www.das-labor.org/wiki/Crypto-avr-lib
13. Engels, D., Fan, X., Gong, G., Hu, H., Smith, E.: Ultra-lightweight cryptography for low-cost RFID tags: Hummingbird algorithm and protocol. Technical report, Centre for Applied Cryptographic Research, CACR (2009)
14. Engels, D., Saarinen, M.-J.O., Schweitzer, P., Smith, E.M.: The Hummingbird-2 Lightweight Authenticated Encryption Algorithm. In: Juels, A., Paar, C. (eds.) RFIDSec 2011. LNCS, vol. 7055, pp. 19–31. Springer, Heidelberg (2012)
15. Fan, X., Hu, H., Gong, G., Smith, E., Engels, D.: Lightweight implementation of Hummingbird cryptographic algorithm on 4-bit microcontrollers. In: International Conference for Internet Technology and Secured Transactions 2009, pp. 1–5 (2009)
16. Ganesan, P., Venugopalan, R., Peddabachagari, P., Dean, A.G., Mueller, F., Sichitiu, M.L.: Analyzing and modeling encryption overhead for sensor network nodes. In: Raghavendra, C.S., Sivalingam, K.M., Govindan, R., Ramanathan, P. (eds.) Wireless Sensor Networks and Applications, pp. 151–159. ACM (2003)

17. Goubin, L.: A Refined Power-Analysis Attack on Elliptic Curve Cryptosystems. In: Desmedt, Y.G. (ed.) PKC 2003. LNCS, vol. 2567, pp. 199–210. Springer, Heidelberg (2002)
18. Gura, N., Patel, A., Wander, A., Eberle, H., Shantz, S.C.: Comparing Elliptic Curve Cryptography and RSA on 8-bit CPUs. In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 119–132. Springer, Heidelberg (2004)
19. Hankerson, D., Menezes, A.J., Vanstone, S.: Guide to Elliptic Curve Cryptography. Springer-Verlag New York, Inc., Secaucus (2003)
20. ISO/IEC. 29192-2: Information technology – security techniques – lightweight cryptography – part 2: Block ciphers, http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=56552
21. Johnson, D., Menezes, A., Vanstone, S.: The elliptic curve digital signature algorithm (ECDSA). International Journal of Information Security 1(1), 36–63 (2001)
22. Joye, M., Yen, S.-M.: Optimal left-to-right binary signed-digit recoding. IEEE Trans. Comput. 49, 740–748 (2000)
23. Kaufmann, T., Poschmann, A.: Enabling standardized cryptography on ultra-constrained 4-bit microcontrollers. In: International IEEE Conference on RFID, Orlando, USA (to appear, 2012)
24. Knudsen, L., Leander, G., Poschmann, A., Robshaw, M.J.B.: PRINTcipher: A Block Cipher for IC-Printing. In: Mangard, S., Standaert, F.-X. (eds.) CHES 2010. LNCS, vol. 6225, pp. 16–32. Springer, Heidelberg (2010)
25. Knuth, D.E.: The art of computer programming, 3rd edn., vol. 2. Addison-Wesley (1997)
26. Koblitz, N.: Elliptic curve cryptosystems. Mathematics of computation 48(177), 203–209 (1987)
27. Kocher, P.C.: 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)
28. Kocher, P.C., Jaffe, J., Jun, B.: Differential Power Analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999)
29. Longa, P., Miri, A.: Fast and flexible elliptic curve point arithmetic over prime fields. IEEE Trans. Comput. 57, 289–302 (2008)
30. Mamiya, H., Miyaji, A., Morimoto, H.: Efficient Countermeasures against RPA, DPA, and SPA. In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 343–356. Springer, Heidelberg (2004)
31. McCurley, K.: A fast portable implementation of the secure hash algorithm, III (July 1994), http://www.mccurley.org
32. Miller, V.S.: Use of Elliptic Curves in Cryptography. In: Williams, H.C. (ed.) CRYPTO 1985. LNCS, vol. 218, pp. 417–426. Springer, Heidelberg (1986)
33. Montgomery, P.L.: Modular multiplication without trial division. Mathematics of Computation 44(170), 519–521 (1985)
34. National Institute of Standards and Technology. FIPS 197: Announcing the advanced encryption standard (AES) (November 2001), http://csrc.nist.gov
35. National Institute of Standards and Technology. FIPS 180-3: Secure hash standard (October 2008), http://csrc.nist.gov
36. Reitwiesner, G.W.: Binary arithmetic. Advances in Computers 1, 231–308 (1960)
37. Rinne, S., Eisenbarth, T., Paar, C.: Performance analysis of contemporary lightweight block ciphers on 8-bit microcontrollers. In: ecrypt workshop SPEED - Software Performance Enhancement for Encryption and Decryption (2007)

38. Rohde, S., Eisenbarth, T., Dahmen, E., Buchmann, J., Paar, C.: Fast Hash-Based Signatures on Constrained Devices. In: Grimaud, G., Standaert, F.-X. (eds.) CARDIS 2008. LNCS, vol. 5189, pp. 104–117. Springer, Heidelberg (2008)
39. Saarinen, M.-J.O.: Cryptanalysis of Hummingbird-1. In: Joux, A. (ed.) FSE 2011. LNCS, vol. 6733, pp. 328–341. Springer, Heidelberg (2011)
40. Schroeppel, R., Orman, H., O'Malley, S., Spatscheck, O.: Fast Key Exchange with Elliptic Curve Systems. In: Coppersmith, D. (ed.) CRYPTO 1995. LNCS, vol. 963, pp. 43–56. Springer, Heidelberg (1995)
41. Seiko Epson Corporation. CMOS 4-bit single chip microcomputer S1C63000 core CPU manual (2011), http://www.epson.jp/device/semicon_e/product/index_mcu.htm
42. Seiko Epson Corporation. CMOS 4-bit single chip microcontroller S1C63003/004/008/016 technical manual (2011), http://www.epson.jp/device/semicon_e/product/index_mcu.htm
43. Seiko Epson Corporation. Microcontrollers (2011), http://www.epsondevice.com/webapp/docs_ic/DownloadServlet?id=ID000463
44. Seiko Epson Corporation. Program development process (2011), http://www.epson.jp/device/semicon_e/product/mcu/development/tool.htm
45. TEMIC Semiconductors. Automotive safety and convenience data book (1996), http://pe2bz.philpem.me.uk/pdf%20on%20typenumber/S/SAFE96.pdf
46. Texas Instruments. CC1010 datasheet (September 2009), http://www.ti.com/lit/ds/symlink/cc1010.pdf
47. Turley, J.: Microprocessors for consumer electronics, PDAs, and communications. Embedded Systems Programming 10, 116–128 (1997)
48. Turley, J.: The Essential guide to semiconductors. Prentice Hall PTR (2003)
49. Uhsadel, L., Poschmann, A., Paar, C.: Enabling Full-Size Public-Key Algorithms on 8-Bit Sensor Nodes. In: Stajano, F., Meadows, C., Capkun, S., Moore, T. (eds.) ESAS 2007. LNCS, vol. 4572, pp. 73–86. Springer, Heidelberg (2007)
50. Vogt, M., Poschmann, A., Paar, C.: Cryptography is feasible on 4-bit microcontrollers - a proof of concept. In: International IEEE Conference on RFID, Orlando, USA, pp. 267–274 (2009)

All Subkeys Recovery Attack on Block Ciphers: Extending Meet-in-the-Middle Approach

Takanori Isobe and Kyoji Shibutani

Sony Corporation
1-7-1 Konan, Minato-ku, Tokyo 108-0075, Japan
{Takanori.Isobe,Kyoji.Shibutani}@jp.sony.com

Abstract. We revisit meet-in-the-middle (MITM) attacks on block ciphers. Despite recent significant improvements of the MITM attack, its application is still restrictive. In other words, most of the recent MITM attacks work only on block ciphers consisting of a bit permutation based key schedule such as KTANTAN, GOST, IDEA, XTEA, LED and Piccolo. In this paper, we extend the MITM attack so that it can be applied to a wider class of block ciphers. In our approach, MITM attacks on block ciphers consisting of a complex key schedule can be constructed. We regard all subkeys as independent variables, then transform the game that finds the user-provided key to the game that finds all independent subkeys. We apply our approach called all subkeys recovery (ASR) attack to block ciphers employing a complex key schedule such as CAST-128, SHACAL-2, KATAN, FOX128 and Blowfish, and present the best attacks on them with respect to the number of attacked rounds in literature. Moreover, since our attack is simple and generic, it is applied to the block ciphers consisting of any key schedule functions even if the key schedule is an ideal function.

Keywords: block cipher, meet-in-the-middle attack, key schedule, CAST-128, SHACAL-2, KATAN, FOX128, Blowfish, all subkeys recovery attack.

1 Introduction

Meet-in-the-middle (MITM) attack, originally introduced in [9], is a generic cryptanalytic technique for block ciphers. It was extended to preimage attacks on hash functions, and several novel techniques to extend the attack have been developed [3,14,4,25,5,11]. Then, it has been shown that those advanced techniques are also applied to block ciphers [7,13,6,18].

Since the MITM attack mainly exploits a low key-dependency in a key schedule, it works well for a block cipher having a simple key schedule such as a key schedule based only on a bit permutation. In fact, most of the recent MITM attacks were applied to ciphers having a simple key schedule such as KTANTAN, GOST, IDEA, XTEA, LED and Piccolo [7,13,18,26,15]. However, as far as we know, no results have been known so far for block ciphers consisting of a complex key schedule[1] except for the recent attack on AES [6]. In general, the MITM attack at least requires two sets of neutral key bits, which are parts of secret key bits, to compute two functions independently. For a simple key schedule such as a permutation based key schedule, since each subkey is directly derived from some secret key bits, it is relatively simple to find "good" neutral key bits. Yet, for a cipher equipped with a complex key schedule, finding neutral key bits, which is the core technique of the MITM attack, is likely to be complicated and specific. For instance, the MITM attack on the 8-round reduced AES-128 in [6] looks complicated and specific, i.e., it seems difficult to directly apply their technique to other block ciphers not having the AES key schedule.

In this paper, we extend the MITM attack, then present a generic and simple approach to evaluate the security of ciphers employing a complex key schedule against the MITM attack. The basic concept of our approach is converting the game of finding the user-provided key (or the secret key) to the game of finding all subkeys so that the analysis can be independent from the structure of the key schedule, by regarding all subkeys as independent variables. This simple conversion enables us to apply the MITM attack to a cipher using any key schedule without analyzing the key schedule. We refer this attack as all subkeys recovery (ASR) attack for simplicity, while our approach is a variant of the MITM attack. We first apply the ASR attack to CAST-128, Blowfish and FOX128 in a straightforward way, then show the best attacks on them with respect to the number of attacked rounds in literature. Moreover, to construct more efficient attack, we present how to efficiently find useful state that contains a smaller number of subkey bits by analyzing internal components, then apply it to SHACAL-2 and KATAN family. The attacks presented in this paper are summarized in Table 6 (see also Table 2). We emphasize that our attack can be applied to any block cipher having any key schedule function even if the key schedule is an ideally random function. This implies that our approach gives generic lower bounds on the security of several block ciphers against the MITM attacks.

This paper is organized as follows. Section 2 gives some notations used throughout this paper. The basic concept of the ASR attack is presented in Section 3. The applications of the basic ASR attack to CAST-128 and Blowfish are demonstrated in Section 4. Some advanced techniques and those applications to SHACAL-2 and KATAN family are introduced in Sections 5 and 6, respectively. Section 7 discusses several features of the ASR attack. Finally, we conclude in Section 8.

2 Notation

The following notation will be used throughout this paper:

[1] In this paper, we refer a complex key schedule as a non-permutation based key schedule such as a key schedule having non-linear components.

$a\|\|b$	: Concatenation.
$\|a\|$	: Bit size of a.
$\ggg$ or $\lll$	: Right or left bit rotation in 32-bit word.
$\oplus$	: Bitwise logical exclusive OR (XOR) operation.
$\boxplus$	: Addition modulo 2^{32} operation.
$\boxminus$	: Subtraction modulo 2^{32} operation.

3 All Subkeys Recovery (ASR) Attack

In this section, the basic concept of the all subkeys recovery (ASR) attack is introduced. First, we briefly present the basic concept, then, the detailed procedure of the basic ASR attack is given. Finally, we show an ASR attack on the balanced Feistel network as a concrete example.

3.1 Basic Concept

The previous MITM attack aims to determine the user-provided key by finding good neutral key bits which are parts of the user-provided key. In order to find good neutral key bits, an attacker needs to thoroughly analyze the key schedule. In general, this makes the analysis complex and specific, and it is difficult to evaluate the security of a wide class of block ciphers, including ciphers having a complex key schedule, against the MITM attack.

The basic concept of the ASR attack is to convert the game of finding the user-provided key to the game of finding all subkeys, regarding all subkeys as independent variables. Note that an attacker is able to encrypt/decrypt any plaintexts/ciphertexts by using all subkeys, even if the user-provided key is unknown. In addition, if a key schedule is invertible, then the user-provided key is obtained from all subkeys.

In the ASR attack, analyzing the key schedule is not mandatory, since we only treat subkeys (not secret key). Obviously, this makes analysis simpler than finding good neutral key bits. In fact, the ASR attack depends only on the sizes of the secret key and the round keys, and the structure of the data processing part such as balanced Feistel network and SPN (substitution-permutation-network).

Moreover, in the ASR attack, the underlying key schedule can be treated as an ideal function. In other words, our attack works even if the key schedule is an ideal function. A concrete block cipher employs an weaker key schedule than an ideal one. Thus, the number of attacked rounds may be extended by thoroughly analyzing the key schedule. However, even without analyzing the key schedule, we show several best attacks on several block ciphers in the single-key setting in the following sections. Thanks to the MITM approach, our attack requires extremely low data requirement, though it requires a lot of memory which is the same order as the time complexity. We may employ memoryless collision search [23] to reduce the memory requirements, while it requires a slightly larger computations, i.e., the time complexity is multiplied by a small constant.

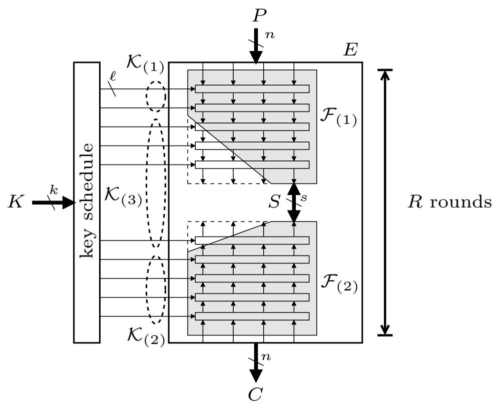


Fig. 1. Basic concept of ASR attack

3.2 Recovering All Subkeys by MITM Approach (Basic Attack)

Let us explain the basic procedure of the ASR attack. As explained in the previous section, we regard all subkeys in the cipher as independent variables. Then, we apply the MITM approach to determine all subkeys that map a plaintext to the ciphertext encrypted by the (unknown) secret key. Suppose that n-bit block cipher E accepting a k-bit secret key K consists of R rounds, and an ℓ-bit subkey is introduced each round (see Fig. 1).

First, an attacker determines an s-bit matching state S. The state S can be computed from a plaintext P and a set of subkey bits $\mathcal{K}_{(1)}$ by a function $\mathcal{F}_{(1)}$ as $S = \mathcal{F}_{(1)}(P, \mathcal{K}_{(1)})$. Similarly, S can be computed from the ciphertext C and another set of subkey bits $\mathcal{K}_{(2)}$ by a function $\mathcal{F}_{(2)}$ as $S = \mathcal{F}_{(2)}^{-1}(C, \mathcal{K}_{(2)})$. $\mathcal{K}_{(3)}$ denotes a set of the remaining subkey bits, i.e., $|\mathcal{K}_{(1)}| + |\mathcal{K}_{(2)}| + |\mathcal{K}_{(3)}| = R \cdot \ell$. By using those $\mathcal{K}_{(1)}$ and $\mathcal{K}_{(2)}$, the attacker can independently compute $\mathcal{F}_{(1)}(P, \mathcal{K}_{(1)})$ and $\mathcal{F}_{(2)}^{-1}(C, \mathcal{K}_{(2)})$. Note that the equation $\mathcal{F}_{(1)}(P, \mathcal{K}_{(1)}) = \mathcal{F}_{(2)}^{-1}(C, \mathcal{K}_{(2)})$ holds when the guessed subkey bits $\mathcal{K}_{(1)}$ and $\mathcal{K}_{(2)}$ are correct. Due to parallel guesses of $\mathcal{K}_{(1)}$ and $\mathcal{K}_{(2)}$, we can efficiently check if the guessed key bits are correct. After this process, it is expected that there will be $2^{R\cdot\ell-s}$ key candidates. Note that the number of key candidates can be reduced by parallel performing the matching with additional plaintext/ciphertext pairs. In fact, using N plaintext/ciphertext pairs, the number of key candidates is reduced to $2^{R\cdot\ell-N\cdot s}$, as long as $N \leq (|\mathcal{K}_{(1)}| + |\mathcal{K}_{(2)}|)/s$. Finally, the attacker exhaustively searches the correct key from the surviving key candidates. The required computations (i.e. the number of encryption function calls) of the attack in total C_{comp} is estimated as

$$C_{comp} = \max(2^{|\mathcal{K}_{(1)}|}, 2^{|\mathcal{K}_{(2)}|}) \times N + 2^{R\cdot\ell-N\cdot s}. \tag{1}$$

The number of required plaintext/ciphertext pairs is $\max(N, \lceil (R\cdot\ell - N\cdot s)/n \rceil)$. The required memory is about $\min(2^{|\mathcal{K}_{(1)}|}, 2^{|\mathcal{K}_{(2)}|}) \times N$ blocks, which is the cost of the table used for the matching. Obviously, the ASR attack works faster than

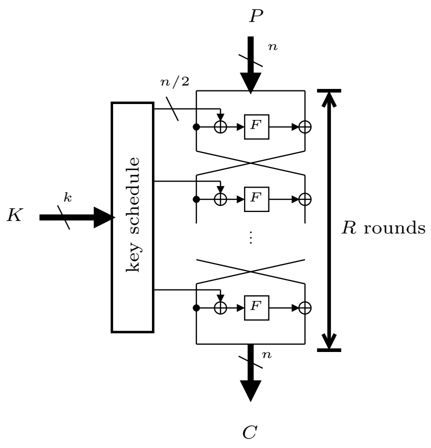


Fig. 2. Balanced Feistel network

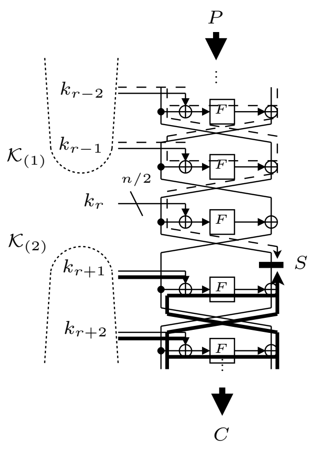


Fig. 3. Partial matching

the brute force attack when Eq.(1) is less than 2^k, which is required computations for the brute force attack. For simplicity, we omit the cost of memory access for finding a match between two lists, assuming that the time complexity of one table look-up is negligible compared to that of one computation of $\mathcal{F}_{(1)}$ or $\mathcal{F}_{(2)}$. The assumption is quite natural in most cases. However, strictly speaking, those costs should be considered. In other words, the cost of $(\max(2^{|\mathcal{K}_{(1)}|}, 2^{|\mathcal{K}_{(2)}|}) \times N)$ memory accesses is added to Eq.(1).

If the number of attacked rounds R and the size of the matching state s are fixed, the time complexity and the memory requirement are dominated by $|\mathcal{K}_{(1)}|$ and $|\mathcal{K}_{(2)}|$. Thus, smaller $|\mathcal{K}_{(1)}|$ and $|\mathcal{K}_{(2)}|$ lead to more efficient attacks with respect to the time and memory complexity. Therefore, the key of the ASR attack is to find the matching state S computed by the smallest $\max(|\mathcal{K}_{(1)}|, |\mathcal{K}_{(2)}|)$.

3.3 ASR Attack on Balanced Feistel Networks

Let us show examples of the ASR attack. Suppose that an example cipher E with n-bit block and k-bit secret key consists of R rounds of the balanced Feistel network as illustrated in Fig. 2. Let the round function F be an $(n/2)$-bit keyed bijective function. Here, for simplicity, we assume that an $(n/2)$-bit subkey is introduced before F each round. Also, k_i denotes the i-th round subkey.

As depicted in Fig. 3, an $(n/2)$-bit state S can be computed independently from $(n/2)$-bit subkey k_r. In other words, S can be computed from P and $\mathcal{K}_{(1)} \in \{k_1, k_2, ..., k_{r-1}\}$. Also, S can be obtained from C and $\mathcal{K}_{(2)} \in \{k_{r+1}, k_{r+2}, ..., k_R\}$.

When $n = k/2$ (e.g., a 128-bit block cipher accepting a 256-bit key), 7 rounds of E can be attacked in a straightforward manner. In this attack, both $\mathcal{F}_{(1)}$ and $\mathcal{F}_{(2)}$ are composed of 3 rounds of E, and thus the sizes of $\mathcal{K}_{(1)}$ and $\mathcal{K}_{(2)}$ are both $3n/2$ bits. As explained in Section 3.2, using six plaintext/ciphertext pairs, the total time complexity C_{comp} is estimated as

$$\begin{aligned} C_{comp} &= \max(2^{|\mathcal{K}_{(1)}|}, 2^{|\mathcal{K}_{(2)}|}) \times N + 2^{R\cdot\ell - N\cdot s} \\ &= 2^{3n/2} \times 6 + 2^{7\cdot n/2 - 6\cdot n/2} \approx 2^{3n/2} \quad (= 2^{3k/4} \ll 2^k) \end{aligned}$$

The required memory is around $6 \times 2^{3n/2}$ blocks. Since C_{comp} is less than $2^{2n}(= 2^k)$, the attack works faster than the exhaustive key search. Note that the number of attacked rounds might be extended by exploiting the subkey relations. Thus, the number of attacked rounds 7 is considered as the lower bounds on the security of this modelled cipher against the ASR attack.

Similarly to this, when $n = k$ (e.g., a 128-bit block cipher accepting a 128-bit key), the attack on at least 3 rounds of E is constructed. In this case, $\mathcal{F}_{(1)}$ and $\mathcal{F}_{(2)}$ consist of 1 round of E, and the sizes of $\mathcal{K}_{(1)}$ and $\mathcal{K}_{(2)}$ are both $n/2$ bits. Therefore, using 3 plaintext/ciphertext pairs (i.e. $N = 3$), the required time complexity C_{comp} is estimated as

$$C_{comp} = \max(2^{|\mathcal{K}_{(1)}|}, 2^{|\mathcal{K}_{(1)}|}) \times N + 2^{R\cdot\ell - N\cdot s} = 2^{n/2} \times 3 + 1 \quad (= 2^{k/2} \times 3 + 1 \ll 2^k).$$

The required memory is around $3 \times 2^{n/2}$ blocks. Consequently, the ASR attack works faster than the brute force attack, which requires about 2^n computations.

Roughly speaking, when Eq.(1) is less than 2^k, the ASR attack works faster than the brute force attack. Therefore, the necessary condition for the basic ASR attack is that the size of subkey is less than the size of secret key.

4 Basic ASR Attacks on CAST-128 and Blowfish

The generic attack on a balanced Feistel network explained in the previous section can be directly applied to a concrete block cipher. In this section, we apply the basic ASR attacks to CAST-128 and Blowfish. In those attacks, the round function F is assumed to be any function even ideal. However, in the case of a concrete cipher, the underlying round function F is specified, i.e., it can be analyzed. By deeply analyzing the round function, the number of attacked rounds may be increased. Such advanced techniques are introduced in the next sections.

The basic parameters of the ciphers analyzed in this paper are listed in Table 1. Table 2 shows the parameters of our (ASR) attacks in this paper.

4.1 Descriptions of CAST-128 and Blowfish

Description of CAST-128. CAST-128 [1,2] is a 64-bit Feistel block cipher accepting a variable key size from 40 up to 128 bits (but only in 8-bit increments). The number of rounds is 16 when the key size is longer than 80 bits. First, the algorithm divides the 64-bit plaintext into two 32-bit words L_0 and R_0, then the i-th round function outputs two 32-bit data L_i and R_i as follows:

$$L_i = R_{i-1}, R_i = L_{i-1} \oplus F_i(R_i, K_i^{rnd}),$$

where F_i denotes the i-th round function and K_i^{rnd} is the i-th round key consisting of a 32-bit masking key K_{m_i} and a 5-bit rotation key K_{r_i}. Each round

Table 1. Basic parameters of our target ciphers

algorithm	block size (n)	key size (k)	subkey size (ℓ)	# rounds (R)
CAST-128 [1]	64	$40 \leq k \leq 128$	37	12 ($k \leq 80$), 16 ($k > 80$)
Blowfish [27]	64	$128 \leq k \leq 448$	32	16
Blowfish-8R* [27]	64	$128 \leq k \leq 192$	32	8
SHACAL-2 [12]	256	$k \leq 512$	32	64
KATAN32/48/64 [8]	32/48/64	80	2	254
FOX128 [16]	128	256	128	16

$*$ Fewer iteration version of Blowfish specified in [27] as a possible simplification.

function F_i consists of four 8 to 32-bit S-boxes, a key dependent rotation, and logical and arithmetic operations (addition, subtraction and XOR). F_i has three variations, and the positions of three logical or arithmetic operations are varied in each round. However, we omit the details of F_i, since, in our analysis, it is regarded as the random function that outputs a 32-bit random value from a 32-bit input R_i and a 37-bit key K_i^{rnd}.

Description of Blowfish. Blowfish [27] is a 16-round Feistel block cipher with 64-bit block and variable key size from 128 up to 448 bit. Several possible simplifications of Blowfish were also described in [27]. We refer one of them that is an 8-round variant (fewer iterations) of Blowfish accepting less than 192 bits of key as Blowfish-8R. First, Blowfish divides a 64-bit plaintext into two 32-bit state L_0 and R_0. Then the i-th round output state $(L_i||R_i)$ is computed as follows:

$$R_i = L_{i-1} \oplus K_i^{rnd}, \quad L_i = R_{i-1} \oplus F(L_i \oplus K_i^{rnd}),$$

where K_i^{rnd} is a 32-bit round key at round i. The F-function F consists of four 8×32 key dependent S-boxes. Note that, in the last round, the 64-bit round key is additionally XORed with the 64-bit state. In this paper, we assume that F-function is known by an attacker, and it is a random 32-bit function. The assumption, i.e., the known F-function setting, has already been used in [29,17].

4.2 ASR Attack on 7-Round Reduced CAST-128

The generic attack on a balanced Feistel network can be directly applied to a variant of CAST-128 which accepts a more than 114 bits key. This variant consists of 16 rounds, since the key size is longer than 80 bits. While the size of each subkey of the CAST-128 is 37 bits including a 32-bit masking key and a 5-bit rotation key, the above explained attack still works faster than the exhaustive key search. For the 7-round reduced CAST-128, we use $|\mathcal{K}_{(1)}| = |\mathcal{K}_{(2)}| = 111(= 37 \times 3)$, since each of $\mathcal{F}_{(1)}$ and $\mathcal{F}_{(2)}$ consists of three rounds of the CAST-128. Consequently, using six plaintext/ciphertext pairs, the total time complexity C_{comp} for the attack on the 7-round reduced CAST-128 is estimated as follows:

Table 2. Parameters of our attacks presented in this paper

algorithm	# attacked rounds	$\lvert\mathcal{K}_{(1)}\rvert$ (forward)	$\lvert\mathcal{K}_{(2)}\rvert$ (backward)	$\lvert\mathcal{K}_{(3)}\rvert$ (remains)	s	attacked key size
CAST-128	7	111	111	37	32	$\mathbf{120} \leq k \leq 128$
Blowfish†	16	256	288	32	32	$\mathbf{292} \leq k \leq 448$
Blowfish-8R†	8	128	160	32	32	$\mathbf{163} \leq k \leq 192$
SHACAL-2	41	484	492	336	4	$\mathbf{485} \leq k \leq 512$
KATAN32	110	68	70	82	1	80
KATAN48	100	71	71	58	1	80
KATAN64	94	71	71	46	1	80
FOX128	5	224	224	192	32	256

† Known F-function setting.

$$\begin{aligned} C_{comp} &= \max(2^{|\mathcal{K}_{(1)}|}, 2^{|\mathcal{K}_{(2)}|}) \times N + 2^{R\cdot\ell - N\cdot s} \\ &= 2^{111} \times 6 + 2^{7\cdot 37 - 6\cdot 32} = 2^{111} \times 6 + 2^{67} \approx 2^{114}. \end{aligned}$$

The number of required known plaintext/ciphertext pairs is only 6 (= max(6, $\lceil(258-6\cdot32)/64\rceil$), and the required memory is about 2^{114} (= $\min(2^{111}, 2^{111}) \times 6$) blocks. Recall that our attack works faster than the exhaustive key search only when the key size of the reduced CAST-128 is more than 114 bits. As far as we know, the previous best attack on the reduced CAST-128 was for only 6 rounds in the single-key setting [31][2]. Thus, surprisingly, this simple attack exceeds the previously best attack on the reduced CAST-128 with respect to the number of attacked rounds.

4.3 ASR Attack on Full Blowfish in Known F-function Setting

In this section, we apply the ASR attack on Blowfish block cipher in the known F-function setting as with [29,17].

For the full (16-round) Blowfish, we choose R_8 as the 32-bit matching state, i.e., $s = 32$. Then $\mathcal{K}_{(1)}$ and $\mathcal{K}_{(2)}$ include 256 (= 32×8) and 288 (= 32×9) bits of subkeys, respectively, and $\mathcal{K}_{(3)} = 32$. When $N = 9$ ($\leq (256 + 288)/32$), the time complexity for finding all subkeys is estimated as

$$C_{comp} = \max(2^{256}, 2^{288}) \times 9 + 2^{576-9\cdot32} = 2^{292}.$$

The number of required data is only 9 (=max(9, $\lceil(576-9\cdot32)/64\rceil$)) known plaintext/ciphertext pairs, and required memory is about 2^{260} (=$\min(2^{256}, 2^{288}) \times 9$) blocks. In this setting, the attack works faster than the exhaustive key search when the key size is more than 292 bits.

Similarly to the attack on the full Blowfish, for the full (8-round) Blowfish-8R, we choose R_4 as the 32-bit matching state, i.e., $s = 32$. Then $\mathcal{K}_{(1)}$ and $\mathcal{K}_{(2)}$

[2] The differential attacks on the 8- and 9-round reduced CAST-128 in weak-key setting were presented in [30].

include 128 (= 32 × 4) and 160 (= 32 × 5) bits of subkeys, respectively, and $\mathcal{K}_{(3)} = 32$. When $N = 5$ ($\leq (128+160)/32$), the time complexity for finding all subkeys is estimated as

$$C_{comp} = \max(2^{128}, 2^{160}) \times 5 + 2^{320-5\cdot 32} = 2^{163}.$$

The number of required data is only 5 (=max(5, $\lceil(320-5\cdot32)/64\rceil$)) known plaintext/ciphertext pairs, and the required memory is about 2^{131} (=min($2^{128}, 2^{160}$)$\times$ 5) blocks. In this setting, the attack works faster than the exhaustive key search when the key size is more than 163 bits.

Note that these attacks are the first results on the full Blowfish with all key classes in the known F-function setting, while the attacks presented in [29,17] work only in the weak key setting, i.e., weak F-functions.

5 Application to SHACAL-2

In this section, we apply the ASR attack to SHACAL-2 block cipher. After a brief description of SHACAL-2, we present the basic ASR attack on the reduced SHACAL-2 without analyzing the internal functions of the cipher. Then, by analyzing the functions of the cipher, we show the matching state that contains fewer bits of subkeys. Finally, we demonstrate an advanced ASR attack by using this matching state. Recall that the basic ASR attack regards internal components such as an F-function of the balanced Feistel network as a black-box function, while the advanced ASR attack analyzes the internal components. An ASR attack on the 5-round reduced FOX128 is presented in Appendix A as another example of the advanced ASR attack.

5.1 Description of SHACAL-2

SHACAL-2 [12] is a 256-bit block cipher based on the compression function of SHA-256 [10]. It was submitted to the NESSIE project and selected to be in the NESSIE portfolio [22].

SHACAL-2 inputs the plaintext to the compression function as the chaining variable, and inputs the key to the compression function as the message block. First, a 256-bit plaintext is divided into eight 32-bit words A_0, B_0, C_0, D_0, E_0, F_0, G_0 and H_0. Then, the state update function updates eight 32-bit variables, A_i, B_i, ..., G_i, H_i in 64 steps as follows:

$$\begin{aligned} T_1 &= H_i \boxplus \Sigma_1(E_i) \boxplus Ch(E_i, F_i, G_i) \boxplus K_i \boxplus W_i, \\ T_2 &= \Sigma_0(A_i) \boxplus Maj(A_i, B_i, C_i), \\ A_{i+1} &= T_1 \boxplus T_2,\ B_{i+1} = A_i,\ C_{i+1} = B_i,\ D_{i+1} = C_i, \\ E_{i+1} &= D_i \boxplus T_1,\ F_{i+1} = E_i,\ G_{i+1} = F_i,\ H_{i+1} = G_i, \end{aligned}$$

where K_i is the i-th step constant, W_i is the i-th step key (32-bit), and the functions Ch, Maj, Σ_0 and Σ_1 are given as follows:

$$\begin{aligned} Ch(X,Y,Z) &= XY \oplus \overline{X}Z, \\ Maj(X,Y,Z) &= XY \oplus YZ \oplus XZ, \\ \Sigma_0(X) &= (X \ggg 2) \oplus (X \ggg 13) \oplus (X \ggg 22), \\ \Sigma_1(X) &= (X \ggg 6) \oplus (X \ggg 11) \oplus (X \ggg 25). \end{aligned}$$

After 64 steps, the function outputs eight 32-bit words A_{64}, B_{64}, C_{64}, D_{64}, E_{64}, F_{64}, G_{64} and H_{64} as the 256-bit ciphertext. Hereafter p_i denotes the i-th step state, i.e., $p_i = A_i||B_i||...||H_i$.

The key schedule of SHACAL-2 takes a variable length key up to 512 bits as the inputs, then outputs 64 32-bit step keys. First, the 512-bit input key is copied to 16 32-bit words W_0, W_1, ..., W_{15}. If the size of the input key is shorter than 512 bits, the key is padded with zeros. Then, the key schedule generates 48 32-bit step keys $(W_{16}, ..., W_{63})$ from the 512-bit key $(W_0, ..., W_{15})$ as follows:

$$W_i = \sigma_1(W_{i-2}) \boxplus W_{i-7} \boxplus \sigma_0(W_{i-15}) \boxplus W_{i-16}, (16 \leq i < 64),$$

where the functions $\sigma_0(X)$ and $\sigma_1(X)$ are defined by

$$\begin{aligned} \sigma_0(X) &= (X \ggg 7) \oplus (X \ggg 18) \oplus (X \gg 3), \\ \sigma_1(X) &= (X \ggg 17) \oplus (X \ggg 19) \oplus (X \gg 10). \end{aligned}$$

5.2 Basic ASR Attack on 37-Step Reduced SHACAL-2

We directly apply the ASR attack described in Section 3 to SHACAL-2. This leads to the attack on the 37-step reduced SHACAL-2.

Due to a generalized Feistel network-like structure of SHACAL-2, the i-step 32-bit word A_i is computed without using subkeys $W_i, W_{i+1}, ..., W_{i+6}$ as mentioned in [14]. Thus, the 15-step state A_{15} can be computed by each of $\mathcal{F}_{(1)}(P, \mathcal{K}_{(1)})$ and $\mathcal{F}_{(2)}^{-1}(C, \mathcal{K}_{(2)})$, where $\mathcal{K}_{(1)} \in \{W_0, W_1, ..., W_{14}\}$ and $\mathcal{K}_{(2)} \in \{W_{22}, ..., W_{36}\}$. Since $|\mathcal{K}_{(1)}| = |\mathcal{K}_{(2)}| = 480 (= 32 \times 15)$ and the size of the matching state A_{15} is 32 bits, by using 22 known plaintext/ciphertext pairs, the time complexity to compute all subkey bits is estimated as

$$C_{comp} = \max(2^{480}, 2^{480}) \times 22 + 2^{37 \cdot 32 - 22 \cdot 32} \approx 2^{485}.$$

The required memory is about 2^{485} blocks. Note that this attack finds a secret key more efficiently than the exhaustive key search only when the key size is more than 485 bits.

Surprisingly, this simple attack exceeds the previous best attack on the reduced SHACAL-2 in the single-key setting for 32 steps [28] with respect to the

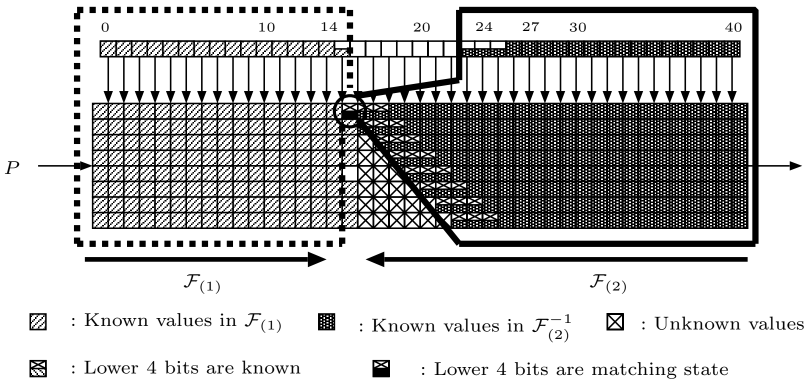

0
10
14
20
24
27
30
40
P
C
$\mathcal{F}_{(1)}$
$\mathcal{F}_{(2)}$
: Known values in $\mathcal{F}_{(1)}$
: Known values in $\mathcal{F}_{(2)}^{-1}$
: Unknown values
: Lower 4 bits are known
: Lower 4 bits are matching state

Fig. 4. Overview of 41 round attack on reduced SHACAL-2

number of attacked rounds [3]. In the following, by deeply analyzing the functions used in SHACAL-2, we show further improvements.

5.3 Advanced ASR Attack on 41-Step Reduced SHACAL-2

In order to extend the basic ASR attack, we choose the lower 4 bits of A_{16} as the matching state. Using this matching state, the 41-step reduced SHACAL-2 can be attacked. The forward and backward functions $\mathcal{F}_{(1)}$ and $\mathcal{F}_{(2)}$ are given as follows (see also Fig. 4).

Forward Computation in $\mathcal{F}_{(1)}$: Due to the structure of SHACAL-2, the lower 4 bits of A_{16} can be computed from the 15-th state p_{15} and the lower 4 bits of W_{15}, since the other bits of W_{15} are not affected to the lower 4 bits of A_{16}. Thus, the matching state S (the lower 4 bits of A_{16}) is calculated as $S = \mathcal{F}_{(1)}(P, \mathcal{K}_{(1)})$, where $\mathcal{K}_{(1)} \in \{W_0, W_1, ..., W_{14}$, the lower 4 bits of $W_{15}\}$ and $|\mathcal{K}_{(1)}| = 484(= 32 \times 15 + 4)$.

Backward Computation in $\mathcal{F}_{(2)}$: We give the following observation.

Observation 1. *The lower t bits of A_{j-10} are obtained from the j-th state p_j and the lower t bits of three subkeys W_{j-1}, W_{j-2} and W_{j-3}.*

[3] The MITM preimage attacks on the reduced SHA-256 were proposed in [5,19]. However, these attacks can not be directly applied to SHACAL-2, because in the block cipher setting, these attacks require the code book, i.e., they require all plaintext/ciphertext pairs. Also, due to those high time complexity, they do not seem to work faster than the exhaustive key search.

In the backward computation, i.e., the inverse step function, the $(j-1)$-th step state p_{j-1} is computed from the j-th step state p_j as follows:

$$\begin{aligned} H_{j-1} &= A_j \boxminus \Sigma_0(B_j) \boxminus Maj(B_j, C_j, D_j) \boxminus \Sigma_1(F_j) \\ &\qquad \boxminus Ch(F_j, G_j, H_j) \boxminus K_{j-1} \boxminus W_{j-1}, \\ D_{j-1} &= E_j \boxminus A_j \boxplus \Sigma_0(B_j) \boxplus Maj(B_j, C_j, D_j), \\ G_{j-1} &= H_j, F_{j-1} = G_j, E_{j-1} = F_j, C_{j-1} = D_j, B_{j-1} = C_j, A_{j-1} = B_j. \end{aligned}$$

Thus, p_{j-1} except for H_{j-1} is obtained from p_j. The lower t bits of H_{j-1} can be computed from p_j and the lower t bits of W_{j-1}. Similarly, from $A_{j-1}, ..., G_{j-1}$ and the lower t bits of H_{j-1} and W_{j-2}, we can compute $A_{j-2}, ..., F_{j-2}$ and the lower t bits of G_{j-2} and H_{j-2}. Furthermore, $A_{j-3}, ..., E_{j-3}$ and the lower t bits of F_{j-3} and G_{j-3} are computed from $A_{j-2}, ..., F_{j-2}$ and the lower t bits of G_{j-2}, H_{j-2} and W_{j-3}. Again, as mentioned in [14], A_{j-10} is determined by p_{j-3} without $W_{j-10}, ..., W_{j-16}$. This relation can be translated to "the lower t bits of A_{j-10} are determined from only $A_{j-3}, ..., E_{j-3}$ and the lower t bits of F_{j-3}, G_{j-3} and H_{j-3}". Therefore, A_{j-10} can be obtained from p_j and the lower t bits of W_{j-1}, W_{j-2} and W_{j-3}.

From Observation 1, the matching state S (the lower 4 bits of A_{16}) can be computed as $S = \mathcal{F}_{(2)}^{-1}(C, \mathcal{K}_{(2)})$, where $\mathcal{K}_{(2)} \in \{W_{26}, ..., W_{40}$, the lower 4 bits of W_{23}, W_{24} and $W_{25}\}$. Thus, $|\mathcal{K}_{(2)}| = 492 (= 32 \times 15 + 4 \times 3)$.

Evaluation. Recall that the matching state S is the lower 4 bits of A_{16}, $|\mathcal{K}_{(1)}| = 484$, $|\mathcal{K}_{(2)}| = 492$ and $|\mathcal{K}_{(3)}| = 336$ $(= 32 \times 7 + (32-4) \times 4)$. Thus, using 244 known plaintext/ciphertext pairs (i.e. $N = 244 \leq (484 + 492)/4$), the time complexity for finding all subkeys is estimated as

$$C_{comp} = \max(2^{484}, 2^{492}) \times 244 + 2^{1312-244\cdot 4} = 2^{500}.$$

The number of required data is 244 (=max(244, $\lceil(1312 - 244 \cdot 4)/256\rceil$)) known plaintext/ciphertext pairs. The memory size is 2^{492} (=min$(2^{484}, 2^{492}) \times 244$) blocks. The attack works more efficiently than the exhaustive key search when the key size is more than 500 bits.

6 Application to KATAN Family

In this section, we analyze KATAN family including KATAN32, KATAN48 and KATAN64. First, we briefly describe the specification of KATAN family. Then, we explain our attack strategy for finding the matching state depending on fewer key bits. Finally, we develop ASR attacks on the reduced KATAN32/48/64. We emphasize that all of our attacks on the reduced KATAN presented in this section are the best (exponential-advantage) attacks in the single key setting with respect to the number of attacked rounds.

Table 3. Parameters of KATAN family

Algorithm	$\lvert L_1\rvert$	$\lvert L_2\rvert$	x_1	x_2	x_3	x_4	x_5	y_1	y_2	y_3	y_4	y_5	y_6
KATAN32	13	19	12	7	8	5	3	18	7	12	10	8	3
KATAN48	19	29	18	12	15	7	6	28	19	21	13	15	6
KATAN64	25	39	24	15	20	11	9	38	25	33	21	14	9

6.1 Description of KATAN

KATAN [8] family is a feedback shift register-based block cipher consisting of three variants: KATAN32, KATAN48 and KATAN64 whose block sizes are 32-, 48- and 64-bit, respectively. All of the KATAN ciphers use the same key schedule accepting an 80-bit key and 254 rounds. The plaintext is loaded into two shift registers L_1 and L_2. Each round, L_1 and L_2 are shifted by one bit, and the least significant bits of L_1 and L_2 are updated by $f_b(L_2)$ and $f_a(L_1)$, respectively. The bit functions f_a and f_b are defined as follows:

$$f_a(L_1) = L_1[x_1] \oplus L_1[x_2] \oplus (L_1[x_3] \cdot L_1[x_4]) \oplus (L_1[x_5] \cdot IR) \oplus k_{2i},$$
$$f_b(L_2) = L_2[y_1] \oplus L_2[y_2] \oplus (L_2[y_3] \cdot L_2[y_4]) \oplus (L_2[y_5] \cdot L_2[y_6]) \oplus k_{2i+1},$$

where $L[x]$ denotes the x-th bit of L, IR denotes the round constant, and k_{2i} and k_{2i+1} denote the 2-bit i-th round key. Note that for KATAN family, the round number starts from 0 instead of 1, i.e., KATAN family consists of round functions starting from the 0-th round to the 253-th round. L_1^i or L_2^i denote the i-th round registers L_1 or L_2, respectively. For KATAN48 or KATAN64, in each round, the above procedure is iterated twice or three times, respectively. All of the parameters for the KATAN ciphers are listed in Table 3.

The key schedule of KATAN ciphers copies the 80-bit user-provided key to $k_0, ..., k_{79}$, where $k_i \in \{0, 1\}$. Then, the remaining 428 bits of the round keys are generated as follows:

$$k_i = k_{i-80} \oplus k_{i-61} \oplus k_{i-50} \oplus k_{i-13} \quad \text{for } i = 80, ..., 507.$$

6.2 Attack Strategy

Recall that the key of our attack is to find the state that contains as small number of subkey bits as possible. In order to find such states, we exhaustively observe the number of key bits involved in each state per round. A pseudo code for counting the number of subkey bits involved in the forward direction for KATAN32 is described in Algorithm 1. We use similar code to observe how many subkey bits are affected to each state of KATAN32 in the backward direction. Also, similar codes are used for counting the number of subkey bits related to each state of KATAN48 and KATAN64. As an example, Table 4 shows the results obtained by this algorithm when $R = 63$ of KATAN32 in the forward direction.

Algorithm 1. Counting the number of key bits involved in each state

```
Require: $R$ /* Evaluated number of rounds */
Ensure: $\mathcal{LK}_1[0], \ldots, \mathcal{LK}_1[|L_1|-1]$ and $\mathcal{LK}_2[0], \ldots, \mathcal{LK}_2[|L_2|-1]$ /* The number of
    key bits involved in each state after $R$ round */
 1: $\mathcal{LK}_1[i] \leftarrow 0$ for $i = 0, \ldots, |L_1|-1$
 2: $\mathcal{LK}_2[i] \leftarrow 0$ for $i = 0, \ldots, |L_2|-1$
 3: for $i = 0$ to $R-1$ do
 4:    for $j = 0$ to $|L_1|-2$ do
 5:       $\mathcal{LK}_1[(|L_1|-1)-j] \leftarrow \mathcal{LK}_1[(|L_1|-1)-j-1]$
 6:    end for
 7:    for $j = 0$ to $|L_2|-2$ do
 8:       $\mathcal{LK}_2[(|L_2|-1)-j] \leftarrow \mathcal{LK}_2[(|L_2|-1)-j-1]$
 9:    end for
10:    $\mathcal{LK}_1[0] \leftarrow \mathcal{LK}_2[y_1] + \mathcal{LK}_2[y_2] + \mathcal{LK}_2[y_3] + \mathcal{LK}_2[y_4] + \mathcal{LK}_2[y_5] + \mathcal{LK}_2[y_6] + 1$
11:    $\mathcal{LK}_2[0] \leftarrow \mathcal{LK}_1[x_1] + \mathcal{LK}_1[x_2] + \mathcal{LK}_1[x_3] + \mathcal{LK}_1[x_4] + (\mathcal{LK}_1[x_5] \cdot IR) + 1$
12: end for
13: return $\mathcal{LK}_1[0], \ldots, \mathcal{LK}_1[|L_1|-1]$ and $\mathcal{LK}_2[0], \ldots, \mathcal{LK}_2[|L_2|-1]$
```

6.3 ASR Attack on 110-Round Reduced KATAN32

We consider the 110-round variant of KATAN32 starting from the first (0-th) round. In this attack, $L_2^{63}[18]$ is chosen as the matching state.

Forward Computation in $\mathcal{F}_{(1)}$: As shown in Table 4, $L_2^{63}[18]$ depends on 68 subkey bits. This implies that $L_2^{63}[18]$ can be computed by a plaintext P and 68 bits of subkeys. More specifically, $L_2^{63}[18] = \mathcal{F}_{(1)}(P, \mathcal{K}_{(1)})$, where $\mathcal{K}_{(1)} \in \{k_0, ..., k_{54}, k_{56}, k_{57}, k_{58}, k_{60}, ..., k_{64}, k_{68}, k_{71}, k_{73}, k_{77}, k_{88}\}$ and $|\mathcal{K}_{(1)}| = 68$.

Backward Computation in $\mathcal{F}_{(2)}$: Table 5 shows the result obtained by Algorithm 1 modified to backward direction on KATAN32 with $R = 47$ starting from 110 round. In the backward computation, the matching state $L_2^{63}[18]$ is computed as $L_2^{63}[18] = \mathcal{F}_{(2)}^{-1}(C, \mathcal{K}_{(2)})$, where $\mathcal{K}_{(2)} \in \{k_{126}, k_{138}, k_{142}, k_{146}, k_{148}, k_{150}, k_{153}, k_{154}, k_{156}, k_{158}, k_{160}, \ldots k_{219}\}$, and $|\mathcal{K}_{(2)}| = 70$.

Evaluation. For the 110-round reduced KATAN32, the matching state S is chosen as $L_2^{63}[18]$ (1-bit state). Since $|\mathcal{K}_{(3)}| = 82 (= 2 \times 110 - 68 - 70)$ which is more than 80 bits, we first determine only $\mathcal{K}_{(1)}$ and $\mathcal{K}_{(2)}$. After that, we additionally mount the MITM approach in order to determine the remaining 82 bits of subkeys.

When $N = 138$ ($\leq (68+70)/1$), the time complexity for finding $\mathcal{K}_{(1)}$ and $\mathcal{K}_{(2)}$ is estimated as

$$C_{comp} = \max(2^{68}, 2^{70}) \times 138 + 2^{138-138\cdot 1} = 2^{77.1}.$$

The number of required data is 138 (=max(138, $\lceil(138 - 138 \cdot 1)/32\rceil$)) known plaintext/ciphertext pairs. The required memory size is about $2^{75.1}$ (=min(2^{68}, 2^{70}) $\times 138$) blocks.

Table 4. Results on KATAN32 with $R = 63$ in forward direction (starting round = 0)

$\mathcal{LK}_1[i]$	0	1	2	3	4	5	6	7	8	9	10	11	12						
# key bits	108	104	102	100	98	98	96	94	92	88	86	84	84						
$\mathcal{LK}_2[i]$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	**18**
# key bits	104	102	100	98	100	93	92	90	88	90	87	85	83	77	75	75	75	74	**68**

Table 5. Results on KATAN32 with $R = 47$ in backward direction (starting round = 109)

$\mathcal{LK}_1[i]$	0	1	2	3	4	5	6	7	8	9	10	11	12						
# key bits	44	46	48	50	52	54	56	58	60	62	64	66	68						
$\mathcal{LK}_2[i]$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	**18**
# key bits	34	36	38	40	42	44	46	48	50	52	54	56	58	60	62	64	66	68	**70**

Finally, we need to find the remaining 82 bits of subkeys by using the simple MITM approach in the setting where $\mathcal{K}_{(1)}$ and $\mathcal{K}_{(2)}$ are known. The required complexity and memory for this process is roughly estimated as 2^{41}. These costs are obviously much less than those of finding $\mathcal{K}_{(1)}$ and $\mathcal{K}_{(2)}$.

6.4 ASR Attack on 100-Round Reduced KATAN48

We consider the 100-round variant of KATAN48 starting from the first (0-th) round. In this attack, $L_2^{58}[28]$ is chosen as the matching state.

In the forward computation, $L_2^{58}[28]$ depends on 71 key bits, namely $L_2^{58}[28] = \mathcal{F}_{(1)}(P, \mathcal{K}_{(1)})$, where $\mathcal{K}_{(1)} \in \{k_0, \dots k_{60}, k_{62}, k_{64}, k_{65}, k_{66}, k_{69}, k_{71}, k_{72}, k_{75}, k_{79}, k_{86}\}$, and $|\mathcal{K}_{(1)}| = 71$. In the backward computation, $L_2^{58}[28]$ depends on 71 key bits, namely $L_2^{58}[28] = \mathcal{F}_{(2)}^{-1}(C, \mathcal{K}_{(2)})$, where $\mathcal{K}_{(2)} \in \{k_{116}, k_{122}, k_{124}, k_{128}, k_{130}, k_{132}, k_{134}, k_{135}, k_{136}, k_{138}, \dots k_{199}\}$, and $|\mathcal{K}_{(2)}| = 71$. Since the size of matching state s is 1, $|\mathcal{K}_{(1)}| = |\mathcal{K}_{(2)}| = 71$ and $|\mathcal{K}_{(3)}| = 58 (= 2 \times 100 - 71 - 71)$, by using $N = 128$ $(\le (71 + 71)/1)$ known plaintext/ciphertext pairs, the time complexity for finding all subkeys is estimated as

$$C_{comp} = \max(2^{71}, 2^{71}) \times 128 + 2^{200-128\cdot 1} = 2^{78}.$$

The number of required data is 128 (=max(128, $\lceil (200 - 128 \cdot 1)/48 \rceil$)) known plaintext/ciphertext pairs. The memory size is 2^{78} (=min($2^{71}, 2^{71}$)$\times 128$) blocks.

6.5 ASR Attack on 94-Round Reduced KATAN64

Similarly to the attack on the 100-round reduced KATAN48, we consider the 94-round variant of KATAN64 starting from the first (0-th) round. In this attack, $L_2^{54}[38]$ is chosen as the 1-bit matching state.

Table 6. Summary of the attacks in the single-key setting

algorithm	# attacked rounds	time	memory [block]	data	reference
CAST-128	6	$2^{88.51}$	Not given	$2^{53.96}$ KP	[31]
	7	2^{114}	2^{114}	6 KP	this paper (Section 4.2)
Blowfish*1	4	-	-	2^{21} CP	[24]
Blowfish†	8	-	-	2^{48} CP	[29]
	16	2^{292}	2^{260}	9 KP	this paper (Section 4.3)
Blowfish-8R†	8	2^{160}	2^{131}	5 KP	this paper (Section 4.3)
SHACAL-2	32	$2^{504.2}$	$2^{48.4}$	$2^{43.4}$ CP	[28]
	41	2^{500}	2^{492}	244 KP	this paper (Section 5)
KATAN32*2	78	2^{76}	Not given	2^{16} CP	[21]
	110	2^{77}	$2^{75.1}$	138 KP	this paper (Section 6.3)
KATAN48*2	70	2^{78}	Not given	2^{31} CP	[21]
	100	2^{78}	2^{78}	128 KP	this paper (Section 6.4)
KATAN64*2	68	2^{78}	Not given	2^{32} CP	[21]
	94	$2^{77.68}$	$2^{77.68}$	116 KP	this paper (Section 6.5)
FOX128	5	$2^{205.6}$	Not given	2^{9} CP	[32]
	5	2^{228}	2^{228}	14 KP	this paper (Appendix A)

† Known F-function setting.

∗1 The attacks on the full Blowfish in the weak key setting were presented in [29] and [17].

∗2 The accelerating key search techniques for the full KATAN32/48/64 were presented in [20].

In the forward computation, $L_2^{54}[38]$ depends on 71 subkey bits, namely $L_2^{54}[38] = \mathcal{F}_{(1)}(P, \mathcal{K}_{(1)})$, where $\mathcal{K}_{(1)} \in \{k_0, \dots k_{61}, k_{63}, k_{64}, k_{65}, k_{66}, k_{68}, k_{69}, k_{71}, k_{75}, k_{82}\}$, and $|\mathcal{K}_{(1)}| = 71$. In the backward computation, $L_2^{54}[38]$ depends on 71 subkey bits, namely $L_2^{54}[38] = \mathcal{F}_{(2)}^{-1}(C, \mathcal{K}_{(2)})$, where $\mathcal{K}_{(2)} \in \{k_{108}, k_{110}, k_{114}, k_{116}, k_{118}, k_{120}, k_{122}, k_{124}, \dots k_{187}\}$, and $|\mathcal{K}_{(2)}| = 71$. Since $s = 1$, $|\mathcal{K}_{(1)}| = |\mathcal{K}_{(2)}| = 71$, and $|\mathcal{K}_{(3)}| = 46 (= 2 \times 94 - 71 - 71)$, by using $N = 116$ $(\leq (71 + 71)/1)$ known plaintext/ciphertext pairs, the time complexity for finding all subkeys is estimated as

$$C_{comp} = \max(2^{71}, 2^{71}) \times 116 + 2^{188-116\cdot 1} = 2^{77.68}.$$

The number of required data is 116 (=max$(116, \lceil (188 - 116 \cdot 1)/64 \rceil)$) known plaintext/ciphertext pairs. The memory size is $2^{77.68}$ (=min$(2^{71}, 2^{71}) \times 116$) blocks.

7 Discussion

On the ASR attack, an attacker attempts to recover all subkeys instead of the user-provided key, regarding all subkeys as independent variables. Note that, if there is no equivalent key, which is a reasonable assumption for a moderate block cipher, there obviously exist unique subkeys that map a given plaintext to the ciphertext encrypted by a secret key. In the standard MITM attack, determining neutral key bits and constructing initial structure called bicliques seem two of the most complicated and important parts in the attack process. However, in our attack, those two procedures are not required, since the attacker focuses only on subkeys and all subkeys are treated equally. Moreover, in the ASR attack, it is not mandatory to analyze the underlying key schedule, since it basically focuses only on the data processing part. These features make the attack simple and generic. While the ASR attack is simple and generic as explained, it is still powerful attack. Indeed, we can significantly improve the previous results on several block ciphers as summarized in Table 6 (see also Table 2 for the details of the target ciphers). We emphasize that our approach is not polynomial-advantage attack such as [20], which requires access of all possible keys, but exponential-advantage attack. Moreover, our attack works on the block ciphers using any key schedule functions even if it is ideal.

While all subkeys are regarded as independent variables in the ASR attack, there must exist some relations between them in an actual block cipher. Thus, if an attacker exploits some properties in the underlying key schedule, the attacker may be able to enhance the attacks presented in this paper. For instance, the following techniques might be useful:

– Finding the user-provided key from the part of subkeys.
– Reducing the search space of subkeys by using relation of subkeys.

Since the purpose of this paper provides generic approach to evaluate the security of block ciphers, we do not show these specific and dedicated techniques. However, by using such techniques, the number of attacked rounds might be increased.

8 Conclusion

We have proposed a new but simple and generic attack on block ciphers. The proposed attack called all subkeys recovery (ASR) attack is based on the meet-in-the-middle (MITM) attack. The previous MITM attack applied to several block ciphers consisting of a simple key schedule such as a permutation based key schedule, since the MITM attack mainly exploits the weakness in the key schedule. However, there have been a few results on the block ciphers having complex key schedule.

In this paper, we applied the ASR attack to several block ciphers employing complex key schedule, regarding all subkeys as independent variables. We showed the ASR attacks on the 7-, 41-, 110-, 100-, 94- and 5-round reduced CAST-128, SHACAL-2, KATAN32, KATAN48, KATAN64 and FOX128, respectively. Moreover, we presented the ASR attacks on the full Blowfish in the known F-function setting. All of our results except for the attack on the reduced FOX128 significantly improved the previous results with respect to the number of attacked rounds.

Acknowledgments. The authors would like to thank to the anonymous referees for their fruitful comments and suggestions.

References

1. Adams, C.: The CAST-128 encryption algorithm. RFC-2144 (May 1997)
2. Adams, C.: Constructing symmetric ciphers using the CAST design procedure. Des. Codes Cryptography 12(3), 283–316 (1997)
3. Aoki, K., Sasaki, Y.: Preimage Attacks on One-Block MD4, 63-Step MD5 and More. In: Avanzi, R.M., Keliher, L., Sica, F. (eds.) SAC 2008. LNCS, vol. 5381, pp. 103–119. Springer, Heidelberg (2009)
4. Aoki, K., Sasaki, Y.: Meet-in-the-Middle Preimage Attacks Against Reduced SHA-0 and SHA-1. In: Halevi, S. (ed.) CRYPTO 2009. LNCS, vol. 5677, pp. 70–89. Springer, Heidelberg (2009)
5. Aoki, K., Guo, J., Matusiewicz, K., Sasaki, Y., Wang, L.: Preimages for Step-Reduced SHA-2. In: Matsui, M. (ed.) ASIACRYPT 2009. LNCS, vol. 5912, pp. 578–597. Springer, Heidelberg (2009)
6. Bogdanov, A., Khovratovich, D., Rechberger, C.: Biclique Cryptanalysis of the Full AES. In: Lee, D.H. (ed.) ASIACRYPT 2011. LNCS, vol. 7073, pp. 344–371. Springer, Heidelberg (2011)
7. Bogdanov, A., Rechberger, C.: A 3-Subset Meet-in-the-Middle Attack: Cryptanalysis of the Lightweight Block Cipher KTANTAN. In: Biryukov, A., Gong, G., Stinson, D.R. (eds.) SAC 2010. LNCS, vol. 6544, pp. 229–240. Springer, Heidelberg (2011)
8. De Cannière, C., Dunkelman, O., Knežević, M.: KATAN and KTANTAN — A Family of Small and Efficient Hardware-Oriented Block Ciphers. In: Clavier, C., Gaj, K. (eds.) CHES 2009. LNCS, vol. 5747, pp. 272–288. Springer, Heidelberg (2009)
9. Diffie, W., Hellman, M.E.: Exhaustive cryptanalysis of the NBS Data Encryption Standard. IEEE Computer 10, 74–84 (1977)
10. FIPS: Secure Hash Standard (SHS). Federal Information Processing Standards Publication 180-4
11. Guo, J., Ling, S., Rechberger, C., Wang, H.: Advanced Meet-in-the-Middle Preimage Attacks: First Results on Full Tiger, and Improved Results on MD4 and SHA-2. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 56–75. Springer, Heidelberg (2010)
12. Handschuh, H., Naccache, D.: SHACAL. NESSIE Proposal (updated) (October 2001), https://www.cosic.esat.kuleuven.be/nessie/updatedPhase2Specs/SHACAL/shacal-tweak.zip
13. Isobe, T.: A Single-Key Attack on the Full GOST Block Cipher. In: Joux, A. (ed.) FSE 2011. LNCS, vol. 6733, pp. 290–305. Springer, Heidelberg (2011)
14. Isobe, T., Shibutani, K.: Preimage Attacks on Reduced Tiger and SHA-2. In: Dunkelman, O. (ed.) FSE 2009. LNCS, vol. 5665, pp. 139–155. Springer, Heidelberg (2009)
15. Isobe, T., Shibutani, K.: Security Analysis of the Lightweight Block Ciphers XTEA, LED and Piccolo. In: Susilo, W., Mu, Y., Seberry, J. (eds.) ACISP 2012. LNCS, vol. 7372, pp. 71–86. Springer, Heidelberg (2012)
16. Junod, P., Vaudenay, S.: FOX: A New Family of Block Ciphers. In: Handschuh, H., Hasan, M.A. (eds.) SAC 2004. LNCS, vol. 3357, pp. 114–129. Springer, Heidelberg (2004)

17. Kara, O., Manap, C.: A New Class of Weak Keys for Blowfish. In: Biryukov, A. (ed.) FSE 2007. LNCS, vol. 4593, pp. 167–180. Springer, Heidelberg (2007)
18. Khovratovich, D., Leurent, G., Rechberger, C.: Narrow-Bicliques: Cryptanalysis of Full IDEA. In: Pointcheval, D., Johansson, T. (eds.) EUROCRYPT 2012. LNCS, vol. 7237, pp. 392–410. Springer, Heidelberg (2012)
19. Khovratovich, D., Rechberger, C., Savelieva, A.: Bicliques for Preimages: Attacks on Skein-512 and the SHA-2 Family. In: Canteaut, A. (ed.) FSE 2012. LNCS, vol. 7549, pp. 244–263. Springer, Heidelberg (2012)
20. Knellwolf, S.: Meet-in-the-middle cryptanalysis of KATAN. In: Proceedings of the ECRYPT Workshop on Lightweight Cryptography (2011)
21. Knellwolf, S., Meier, W., Naya-Plasencia, M.: Conditional Differential Cryptanalysis of NLFSR-Based Cryptosystems. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 130–145. Springer, Heidelberg (2010)
22. NESSIE consortium: NESSIE portfolio of recommended cryptographic primitives (2003), https://www.cosic.esat.kuleuven.be/nessie/deliverables/decision-final.pdf
23. van Oorschot, P.C., Wiener, M.J.: Parallel collision search with cryptanalytic applications. J. Cryptology 12(1), 1–28 (1999)
24. Rijmen, V.: Cryptanalysis and design of iterated block ciphers. Doctoral Dissertation, K. U. Leuven (1997)
25. Sasaki, Y., Aoki, K.: Finding Preimages in Full MD5 Faster Than Exhaustive Search. In: Joux, A. (ed.) EUROCRYPT 2009. LNCS, vol. 5479, pp. 134–152. Springer, Heidelberg (2009)
26. Sasaki, Y., Wang, L., Sakai, Y., Sakiyama, K., Ohta, K.: Three-Subset Meet-in-the-Middle Attack on Reduced XTEA. In: Mitrokotsa, A., Vaudenay, S. (eds.) AFRICACRYPT 2012. LNCS, vol. 7374, pp. 138–154. Springer, Heidelberg (2012)
27. Schneier, B.: Description of a New Variable-length Key, 64-bit Block Cipher (Blowfish). In: Anderson, R. (ed.) FSE 1993. LNCS, vol. 809, pp. 191–204. Springer, Heidelberg (1994)
28. Shin, Y., Kim, J.-S., Kim, G., Hong, S.H., Lee, S.-J.: Differential-Linear Type Attacks on Reduced Rounds of SHACAL-2. In: Wang, H., Pieprzyk, J., Varadharajan, V. (eds.) ACISP 2004. LNCS, vol. 3108, pp. 110–122. Springer, Heidelberg (2004)
29. Vaudenay, S.: On the Weak Keys of Blowfish. In: Gollmann, D. (ed.) FSE 1996. LNCS, vol. 1039, pp. 27–32. Springer, Heidelberg (1996)
30. Wang, M., Wang, X., Chow, K.P., Hui, L.C.K.: New differential cryptanalytic results for reduced-round CAST-128. IEICE Transactions 93-A(12), 2744–2754 (2010)
31. Wang, M., Wang, X., Hu, C.: New Linear Cryptanalytic Results of Reduced-Round of CAST-128 and CAST-256. In: Avanzi, R.M., Keliher, L., Sica, F. (eds.) SAC 2008. LNCS, vol. 5381, pp. 429–441. Springer, Heidelberg (2009)
32. Wu, W., Zhang, W., Feng, D.: Integral Cryptanalysis of Reduced FOX Block Cipher. In: Won, D.H., Kim, S. (eds.) ICISC 2005. LNCS, vol. 3935, pp. 229–241. Springer, Heidelberg (2006)

Appendix

A Application to FOX128

We apply the ASR attack to the 5-round reduced FOX128.

A.1 Description of FOX128

FOX128 is a variant of FOX family [16] consisting of a 16-round modified Lai-Massey scheme with 128-bit block and 256-bit key. A 128-bit input state at round i is denoted as four 32-bit words $(LL_{i-1} \mid\mid LR_{i-1} \mid\mid RL_{i-1} \mid\mid RR_{i-1})$. The i-th round function updates the input state using the 128-bit i-th round key K_i^{rnd} as follows:

$$(LL_i||LR_i) = (\texttt{or}(LL_{i-1} \oplus \phi_L)||LR_{i-1} \oplus \phi_L)$$
$$(RL_i||RR_i) = (\texttt{or}(RL_{i-1} \oplus \phi_R)||RR_{i-1} \oplus \phi_R),$$

where `or` denotes a function converting two 16-bit inputs x_0 and x_1 to x_1 and $(x_0 \oplus x_1)$, and $(\phi_L||\phi_R) = \texttt{f64}((LL_{i-1} \oplus LR_{i-1})||(RL_{i-1} \oplus RR_{i-1}), K_i^{rnd})$. `f64` consisting of two 8 8-bit S-box layers `sigma8` separated by the 8×8 MDS matrix `mu8` returns a 64-bit data from a 64-bit input X and two 64-bit subkeys LK_i^{rnd} and RK_i^{rnd} as $(\texttt{sigma8}(\texttt{mu8}(\texttt{sigma8}(X \oplus LK_i^{rnd})) \oplus RK_i^{rnd}) \oplus LK_i^{rnd})$. Two 64-bit subkeys LK_i^{rnd} and RK_i^{rnd} are derived from K_i^{rnd} as $K_i^{rnd} = (LK_i^{rnd}||RK_i^{rnd})$.

A.2 ASR Attack on 5-Round Reduced FOX128

For the 5-round reduced FOX128, the following one-round keyless linear relation can be exploited for the matching:

$$LL_{i+1} \oplus OR^{-1}(LR_{i+1}) = LL_i \oplus LR_i.$$

If we know LL_2 and LR_2, $LL_2 \oplus OR^{-1}(LR_2)$ can be obtained. Thus, we choose LL_2 and LR_2 as the matching state in the forward computation. The 32-bit states LL_2 and LR_2 are computed from a 128-bit subkey K_1^{rnd}, a 64-bit subkey LK_2^{rnd} and the left most 32 bits of RK_2^{rnd}, i.e., $(LL_2, LR_2) = \mathcal{F}_{(1)}(P, \mathcal{K}_{(1)})$, where $\mathcal{K}_{(1)} \in \{K_1^{rnd}, LK_2^{rnd}$, the left most 32 bits of $RK_2^{rnd}\}$, and $|\mathcal{K}_{(1)}| = 224 (= 128 + 64 + 32)$. Similarly, we choose LL_3 and LR_3 as the matching state in the backward computation. Since the similar relation holds in the backward computation, LL_3 and LR_3 are computed as $(LL_3, LR_3) = \mathcal{F}_{(2)}^{-1}(C, \mathcal{K}_{(2)})$, where $\mathcal{K}_{(2)} \in \{K_5^{rnd}, LK_4^{rnd}$, the left most 32 bits of $RK_4^{rnd}\}$, and $|\mathcal{K}_{(2)}| = 224$. Thus, using the parameter $N = 13$ $(\leq (224 + 224)/32)$, the time complexity for finding all round keys is estimated as

$$C_{comp} = \max(2^{224}, 2^{224}) \times 13 + 2^{640-13\cdot 32} = 2^{228}.$$

The number of required data is only 13 (=max(13, $\lceil (640 - 13 \cdot 32)/64 \rceil$)) known plaintext/ciphertext pairs, and required memory is about 2^{228} (=min$(2^{224}, 2^{224}) \times 13$) blocks.

Improved Cryptanalysis of the Block Cipher KASUMI

Keting Jia[1], Leibo Li[2], Christian Rechberger[3], Jiazhe Chen[2], and Xiaoyun Wang[1,3,⋆]

[1] Institute for Advanced Study, Tsinghua University, China
{ktjia,xiaoyunwang}@mail.tsinghua.edu.cn
[2] Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, Shandong University, China
{lileibo,jiazhechen}@mail.sdu.edu.cn
[3] Department of Mathematics, Technical University of Denmark, Denmark
c.rechberger@mat.dtu.dk

Abstract. KASUMI is a block cipher which consists of eight Feistel rounds with a 128-bit key. Proposed more than 10 years ago, the confidentiality and integrity of 3G mobile communications systems depend on the security of KASUMI. In the practically interesting single key setting, only up to 6 rounds have been attacked so far. In this paper we use some observations on the FL and FO functions. Combining these observations with a key schedule weakness, we select some special input and output values to refine the general 5-round impossible differentials and propose the first 7-round attack on KASUMI with time and data complexities similar to the previously best 6-round attacks. This leaves now only a single round of security margin.

The new impossible differential attack on the last 7 rounds needs $2^{114.3}$ encryptions with $2^{52.5}$ chosen plaintexts. For the attack on the first 7 rounds, the data complexity is 2^{62} known plaintexts and the time complexity is $2^{115.8}$ encryptions.

Keywords: KASUMI, Impossible Differential, Cryptanalysis.

1 Introduction

The block cipher KASUMI is designed for 3GPP (3rd Generation Partnership Project, which is the body standardizing the next generation of mobile telephony) as the basis of confidentiality and integrity algorithms by ETSI SAGE [11]. Nowadays, it is widely used in UMTS, GSM and GPRS mobile communications systems [12].

KASUMI has eight Feistel rounds with a 128-bit key optimized for hardware performance, and is a slightly modified version of the block cipher MISTY1 [7]. Because of its importance, KASUMI attracts a great deal of attention of cryptology researchers. Several attacks on variants of KASUMI were published in

[⋆] Corresponding author.

past years [8,9,10]. In the single-key setting, the best result is an impossible differential attack on a 6-round version of the cipher presented by Kühn [5]. In the related-key setting, Blunden and Escott gave a differential attack of KASUMI reduced to 6 rounds [3]. Later, Biham et al. introduced related-key boomerang and rectangle attacks on the full 8-round KASUMI, which need $2^{78.7}$ and $2^{76.1}$ encryptions respectively [2]. At Crypto 2010, Dunkelman et al. proposed a practical related-key attack on the full KASUMI by using a new strategy named sandwich attack [4], which is a formal extension of boomerang attack. However, these attacks assume control over the differences of two or more related keys in all the 128 key bits. This gives not only the attacker a lot more degrees of freedom, it also renders the resulting attack inapplicable in most real-world usage scenarios.

For an impossible differential attack, the secret key is obtained by eliminating wrong keys which bring about the input and output values of the impossible differential. The general 5-round impossible differential $(0, a) \overset{5R}{\nrightarrow} (0, a)$ [1], that holds for any balanced Feistel scheme, was used to attack 6-round KASUMI, where a is a 32-bit non-zero value [5]. We observe that the output difference only depends on 64 bits of the key when the input difference is selected as $(0, *\|0)$. Hence we consider a new, more fine-grained impossible differential $(0, a_l\|0) \overset{5R}{\nrightarrow} (0, a_l\|0)$, where a_l is a 16-bit non-zero value. We mount this impossible differential on round 2 to 6 to analyze the last 7 rounds, and the input and output values of the impossible differential obtained by partial encryption and decryption in the extended rounds only depend on 112 bit keys. The attack costs $2^{52.5}$ chosen plaintexts and $2^{114.3}$ encryptions. Because the positions of the FL and FO functions are different in even rounds and odd rounds, the above impossible differential attack is not applied to the first 7 rounds. However, we have some new observations on the FL function, with which the wrong keys are eliminated earlier than before. The new attack on the first 7 rounds of KASUMI needs 2^{62} known plaintexts and $2^{115.8}$ encryptions. A summary of our attacks and previous attacks with a single key is given in Table 1.

Table 1. Summary of the attacks on KASUMI

Attack Type	Rounds	Data	Time	Source
Higher-Order Differential	5	$2^{22.1}$ CP	$2^{60.7}$ Enc	[10]
Higher-Order Differential	5	$2^{28.9}$ CP	$2^{31.2}$ Enc	[9]
Integral-Interpolation	6	2^{48} CP	$2^{126.2}$ Enc	[8]
Impossible Differential	6	2^{55} CP	2^{100} Enc	[5]
Impossible Differential	7(2-8)	$2^{52.5}$ CP	$2^{114.3}$ Enc	Sect. 4
Impossible Differential	7(1-7)	2^{62} KP	$2^{115.8}$ Enc	Sect. 5

CP refers to the number of chosen plaintexts.
KP refers to the number of known plaintexts.
Enc refers to the number of encryptions.

The paper is organized as follows. We give a brief description of the block cipher KASUMI in Sect. 2. Some observations used in our cryptanalysis are shown in Sect. 3. In Sect. 4, we propose an improved impossible differential attack on the last 7 rounds of KASUMI. And the impossible differential attack on the first 7 rounds of KASUMI is presented in Sect. 5. We summarize the findings and conclude in Sect. 6.

2 Description of KASUMI

KASUMI works on a 64-bit block and uses a 128-bit key. We give a brief description of KASUMI in this section and discuss cost models for evaluating attack complexities.

Key Schedule. In order to make the hardware significantly smaller and reduce key set-up time, the key schedule of KASUMI is much simpler than the original key schedule of MISTY1. The 128-bit key K is divided into eight 16-bit words: k_1,k_2, ..., k_8, i.e., $K = (k_1, k_2, k_3, k_4, k_5, k_6, k_7, k_8)$. In each round, eight key words are used to compute the round subkeys, which are made up of three parts KL_i, KO_i and KI_i. Here, $KL_i = (KL_{i,1}, KL_{i,2})$, $KO_i = (KO_{i,1}, KO_{i,2}, KO_{i,3})$ and $KI_i = (KI_{i,1}$, $KI_{i,2}$, $KI_{i,3})$. We summarize the details of the key schedule of KASUMI in Tab. 2.

Table 2. The key schedule of KASUMI

Round	$KL_{i,1}$	$KL_{i,2}$	$KO_{i,1}$	$KO_{i,2}$	$KO_{i,3}$	$KI_{i,1}$	$KI_{i,2}$	$KI_{i,3}$
1	$k_1 \lll 1$	k'_3	$k_2 \lll 5$	$k_6 \lll 8$	$k_7 \lll 13$	k'_5	k'_4	k'_8
2	$k_2 \lll 1$	k'_4	$k_3 \lll 5$	$k_7 \lll 8$	$k_8 \lll 13$	k'_6	k'_5	k'_1
3	$k_3 \lll 1$	k'_5	$k_4 \lll 5$	$k_8 \lll 8$	$k_1 \lll 13$	k'_7	k'_6	k'_2
4	$k_4 \lll 1$	k'_6	$k_5 \lll 5$	$k_1 \lll 8$	$k_2 \lll 13$	k'_8	k'_7	k'_3
5	$k_5 \lll 1$	k'_7	$k_6 \lll 5$	$k_2 \lll 8$	$k_3 \lll 13$	k'_1	k'_8	k'_4
6	$k_6 \lll 1$	k'_8	$k_7 \lll 5$	$k_3 \lll 8$	$k_4 \lll 13$	k'_2	k'_1	k'_5
7	$k_7 \lll 1$	k'_1	$k_8 \lll 5$	$k_4 \lll 8$	$k_5 \lll 13$	k'_3	k'_2	k'_6
8	$k_8 \lll 1$	k'_2	$k_1 \lll 5$	$k_5 \lll 8$	$k_6 \lll 13$	k'_4	k'_3	k'_7

$x \lll i$: x rotates left by i bits.
$k'_i = k_i \oplus c_i$, where the c_is are fixed constants.

Encryption. KASUMI is a Feistel structure with 8 rounds. Each round is made up of an FL function and an FO function. In odd numbered rounds the FL function precedes the FO function, whereas in even numbered rounds the FO function precedes the FL function. See Fig. 1 (a) for an illustration.

Let $L_{i-1}||R_{i-1}$ be the input of the i-th round, and then the round function is defined as

$$L_i = FO(FL(L_{i-1}, KL_i), KO_i, KI_i) \oplus R_{i-1},$$
$$R_i = L_{i-1},$$

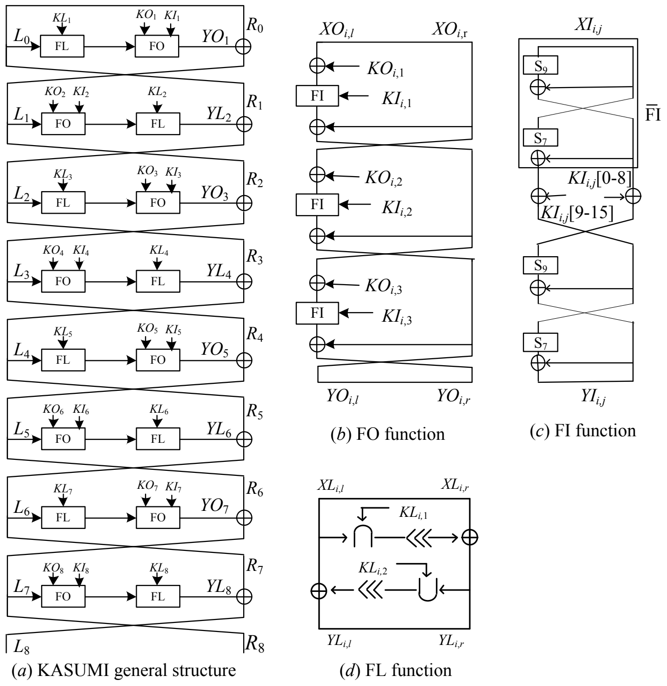


Fig. 1. The structure and building blocks of the block cipher KASUMI

where $i = 1, 3, 5, 7$.'$\oplus$' denotes the bitwise exclusive-or (XOR), and '$\|$' represents the concatenation. When $i = 2, 4, 6, 8$,

$$L_i = FL(FO(L_{i-1}, KO_i, KI_i), KL_i) \oplus R_{i-1},$$
$$R_i = L_{i-1}.$$

Here, $L_0 \| R_0$, $L_8 \| R_8$ are the plaintext and ciphertext respectively, and L_{i-1}, R_{i-1} denote the left and right 32-bit halves of the i-th round input.

The FL function is a simple key-dependent boolean function, depicted in Fig. 1 (c). Let the inputs of the FL function of the i-th round be $XL_i = XL_{i,l} \| XL_{i,r}$, $KL_i = (KL_{i,1}, KL_{i,2})$, the output be $YL_i = YL_{i,l} \| YL_{i,r}$, where $XL_{i,l}, XL_{i,r}, YL_{i,l}$ and $YL_{i,r}$ are 16-bit integers. And the FL function is defined as follows:

$$YL_{i,r} = ((XL_{i,l} \wedge KL_{i,1}) \lll 1) \oplus XL_{i,r}, \quad (1)$$
$$YL_{i,l} = ((YL_{i,r} \vee KL_{i,2}) \lll 1) \oplus XL_{i,l}, \quad (2)$$

where '$\wedge$' and '$\vee$' denote bitwise AND and OR respectively, and '$x \lll i$' implies that x rotates left by i bits. FL_i is the FL function of i-th round with subkey KL_i.

The FO function provides the non-linear property in each round, which is another three-round Feistel structure consisting of three FI functions and key mixing stages. The FO function is depicted in Fig. 1 (b). There is a 96-bit subkey in FO function of each round (48 subkey bits used in the FI functions and 48 subkey bits in the key mixing stages). Let $XO_i = XO_{i,l} \| XO_{i,r}$, $KO_i = (KO_{i,1}, KO_{i,2}, KO_{i,3})$, $KI_i = (KI_{i,1},\ KI_{i,2},\ KI_{i,3})$ be the inputs of the FO function of i-th round, and $YO_i = YO_{i,l} \| YO_{i,r}$ be the corresponding output, where $XO_{i,l}, XO_{i,r}, YO_{i,l}, YO_{i,r}$ and $\overline{XI}_{i,3}$ are 16-bit integers. Then the FO function has the form

$$\begin{aligned}\overline{XI}_{i,3} &= FI((XO_{i,l} \oplus KO_{i,1}), KI_{i,1}) \oplus XO_{i,r},\\ YO_{i,l} &= FI((XO_{i,r} \oplus KO_{i,2}), KI_{i,2}) \oplus \overline{XI}_{i,3},\\ YO_{i,r} &= FI((\overline{XI}_{i,3} \oplus KO_{i,3}), KI_{i,3}) \oplus YO_{i,l}.\end{aligned}$$

For simplicity, define the FO function of i-th round as FO_i.

The FI function uses two sboxes S7 and S9 which are permutations of 7-bit to 7-bit and 9-bit to 9-bit respectively. Suppose the inputs of the j-th FI function of the i-th round are $XI_{i,j}$, $KI_{i,j}$ and the output is $YI_{i,j}$, where $XI_{i,j}$ and $YI_{i,j}$ are 16-bit integers. In order to abbreviate the FI function, we define half of FI function as $\overline{FI}$, which is a 16-bit to 16-bit permutation. The structure of FI and $\overline{FI}$ is depicted in Fig. 1 (d). $\overline{YI}_{i,j} = \overline{FI}(XI_{i,j})$ is defined as

$$\begin{aligned}\overline{YI}_{i,j}[0-8] &= S9(XI_{i,j}[7-15]) \oplus XI_{i,j}[0-6],\\ \overline{YI}_{i,j}[9-15] &= S7(XI_{i,j}[0-6]) \oplus \overline{YI}_{i,j}[0-6],\end{aligned}$$

where $z[i_1 - i_2]$ denotes the $(i_2 - i_1 + 1)$ bits from the i_1-th bit to i_2-th bit of z, and ‘0‘ is the least significant bit. The FI function is simplified as

$$YI_{i,j} = FI(XI_{i,j}, KI_{i,j}) = \overline{FI}((\overline{FI}(XI_{i,j}) \oplus KI_{i,j}) \lll 7).$$

Denote $FI_{i,j}$ as the j-th FI function of the i-th round with subkey $KI_{i,j}$.

3 Some Observations of KASUMI

Let $\Delta X = X \oplus X'$ be the difference of two values X and X'. We describe some observations on the FO and FL functions, which are used in our cryptanalysis of KASUMI.

Observation 1. *Given a pair of input values (XO, XO') of the FO function with difference $\Delta XO = (\Delta XO_l \|\ \Delta XO_r) = (a_l \| 0)$, where a_l is a 16-bit non-zero value. Let ΔYO $(\Delta YO_l \| \Delta YO_r)$ be the corresponding output difference, and then ΔYO only depends on the 64-bit subkey KI_1, KO_1, KI_3 and KO_3.*

Observation 2. *[6] Let X, X' be l-bit values, and $\Delta X = X \oplus X'$. Then there are two difference properties of AND and OR operations, such that*

$$\begin{aligned}(X \wedge K) \oplus (X' \wedge K) &= \Delta X \wedge K,\\ (X \vee K) \oplus (X' \vee K) &= \Delta X \oplus (\Delta X \wedge K).\end{aligned}$$

Observation 3. *Let $a_l \| a_r$ be the input differences of functions FL_1 and FL_7, and the input differences of $FI_{1,2}$ and $FI_{7,2}$ be zero. Then the following equations hold.*

$$(a_l \wedge (k_1 \lll 1)) \lll 1 = a_r, \tag{3}$$
$$(a_l \wedge (k_7 \lll 1)) \lll 1 = a_r. \tag{4}$$

The input differences of $FI_{1,2}$ and $FI_{7,2}$ are zero, so the right 16 bits of output differences of FL_1 and FL_7 are zero. By the definition of the FL function and Observation 2, equations (3) and (4) hold. The following two observations are deduced as well.

Observation 4. *Based on equations (3) and (4), we can get*

$$(a_l \lll 1) \vee \neg a_r = 0xffff. \tag{5}$$

Because the equations (3) and (4) can be represented as 16 parallel equations,

$$\begin{aligned} a_l[j+1] \wedge k_1[j] &= a_r[j+2], \\ a_l[j+1] \wedge k_7[j] &= a_r[j+2], \end{aligned} \qquad j = 0, 1, \ldots, 15. \tag{6}$$

it is obvious that there are only 3 out of 4 values of $(a_l[j+1], a_r[j+2])$ possible, i.e. $(0,0), (1,0), (1,1)$, where $j+1$ and $j+2$ are values mod 16. Therefore we have Observation 4. And the equation (5) holds with probability $(\frac{3}{4})^{16} = 2^{-6.64}$ when both a_l and a_r are uniformly chosen from 2^{16} values. This observation is used to select some special impossible differentials to decrease the time complexity of the key recovery in the attack on the first 7-round KASUMI.

Observation 5. *Suppose $(a_l[j+1],\ a_r[j+2])$ is chosen uniformly from the set $\{(1,1),\ (1,0), (0,0)\}$, where $j = 0, \ldots, 15$, the expected number of the subkey (k_1, k_7) such that the equations (3) and (4) both hold together is 2^{16}.*

For each equation (6), there are 4 values of the subkey $(k_1[j], k_7[j])$ when $(a_l[j+1], a_r[j+2]) = (0,0)$, and there is a value of the subkey $(k_1[j], k_7[j])$ when $(a_l[j+1], a_r[j+2]) = (1,0)$ or $(a_l[j+1], a_r[j+2]) = (1,1)$. Suppose $(a_l[j+1],\ a_r[j+2])$ is chosen uniformly from the set $\{(1,1),\ (1,0), (0,0)\}$, the expected number of the subkey (k_1, k_7) is

$$\sum_{j=1}^{16} \binom{16}{j} \left(\frac{1}{3}\right)^j 4^j \left(\frac{2}{3}\right)^{16-j} = 2^{16}.$$

This observation is used to eliminate wrong keys for the impossible differential attack on the first 7 rounds of KASUMI.

Precomputation. In order to decrease the time complexity, we consider FI as a big sbox, and construct two key dependent difference tables of FI. For all 2^{31} possible input pairs (XI, XI') of the FI function, and 2^{16} possible subkeys KI, compute the corresponding output pairs (YI, YI'). Store the subkey KI and output value YI in a hash table T_1 indexed by 48-bit value $(XI \| XI' \| \Delta YI)$. Then there is one KI for each index on average. Store the value (XI, YI) in a hash table T_2 indexed by 48-bit value $(KI \| \Delta XI \| \Delta YI)$, and then each XI corresponds to an index on average.

4 Impossible Differential Attack on the Last 7 Rounds of KASUMI

The generic 5-round impossible differential of Feistel structure is utilized to analyze 6-round KASUMI, which is: $(0, a) \overset{5R}{\nrightarrow} (0, a)$, where a is a 32-bit non-zero value [5]. Combined with the Feistel structure of the round function, some special values of a are selected to attack the 7-round version of KASUMI.

For the attack on the last 7 rounds, we select the 5-round impossible differential as:

$$(0, a_l \| 0) \overset{5R}{\nrightarrow} (0, a_l \| 0),$$

where a_l is 16-bit non-zero value. The choice of difference $a_l \| 0$ is to minimize the key words guessing when the differential is used to attack the last 7 rounds of KASUMI. We mount the 5-round impossible differential from round 3 to round 7, and extend one round forward and backward, respectively.

Based on observations 1 and 2, we know that, if the input difference of the second round is selected as $\Delta L_1 = (a_l \| 0)$, k_5 and k_7 are not involved in the computation of the output difference $(\Delta L_2, \Delta R_2)$. Similarly, for the backward direction, the input difference of the 8th round $(\Delta L_7, \Delta R_7)$ can be obtained by avoiding guessing (k_3, k_5). Apparently, k_5 does not affect either $(\Delta L_2, \Delta R_2)$ or $(\Delta L_7, \Delta R_7)$, which can help us to reduce the complexity of the attack.

The impossible differential attack on the last 7-round variant of KASUMI is demonstrated as follows, see also Fig. 2.

1. Choose 2^n structures of plaintexts, with each structure containing 2^{48} plaintexts $(L_1, R_1) = (* \| x, * \| *)$, where x is a fixed 16-bit value, '*' takes all the possible 16-bit values. There exist 2^{95} pairs whose input differences are of the form $(* \| 0, * \| *)$ in each structure. Query their corresponding ciphertexts (L_8, R_8) and store (L_1, R_1, L_8, R_8) in a hash table indexed by 32-bit values $L_{1,l} \oplus R_{8,l}$ and $R_{8,r}$. Save the plaintext-ciphertext pair, such that $\Delta L_{1,l} = \Delta R_{8,l}$ and $\Delta R_{8,r} = 0$. There are $2^{n+95-32} = 2^{n+63}$ kept pairs on average.
2. Considering the key schedule and the definition of the round function, the subkey (k_4, k_6, k_7, k_8) can be deduced by guessing the 48-bit subkey (k_1, k_2, k_3). Therefore we guess the subkey (k_1, k_2, k_3) and compute the value $\Delta XL_{8,l} = \Delta YO_{8,l}$ for each plaintext-ciphertext pair. In accordance with the round-function FO, we get $\Delta YI_{8,1} = \Delta YO_{8,l}$. Partially decrypt $(R_{8,l}, R'_{8,l})$ to get the intermediate value $(XI_{8,1}, XI'_{8,1})$. Then obtain the candidate k'_4 by accessing table T_1.

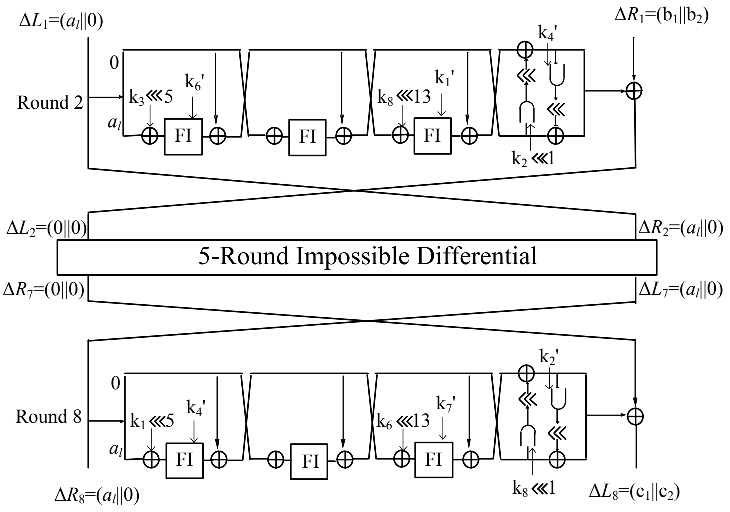


Fig. 2. Impossible differential attack on KASUMI reduced to rounds 2-8

3. Calculate the value $\Delta XL_{2,l} = \Delta YO_{2,l} = \Delta YI_{2,1}$, partially encrypt $(L_{1,l}, L'_{1,l})$ to get the intermediate value $(XI_{2,1}, XI'_{2,1})$, and then search the candidate k'_6 from table T_1.
4. Compute the value $\Delta XL_{2,r}$ $(\Delta YO_{2,r})$, and get the difference of intermediate value $\Delta YI_{2,3} = \Delta YO_{2,l} \oplus \Delta YO_{2,r}$. Since $\Delta XI_{2,3} = \Delta YO_{2,l}$ is known, we can get $(XI_{2,3}, XI'_{2,3})$ by a memory access to hash table T_2. Then partially encrypt $(XI_{2.1}, XI'_{2,1})$ to obtain k_8.
5. From k_8 and Observation 4, we get the value $\Delta XL_{8,r}$ $(\Delta YO_{8,r})$, and then obtain the difference of intermediate value $\Delta YI_{8,3} = \Delta YO_{8,l} \oplus \Delta YO_{8,r}$. Partially encrypt $(XI_{8,1}, XI'_{8,1})$ to get the value $(XI_{8,3}, XI'_{8,3})$, and then obtain the candidate of k'_7 through a memory access to hash table T_1. Thus 16-bit k_7, 48-bit subkey $k_4\|k_6\|k_8$ and 48-bit guessed value $k_1\|k_2\|k_3$ result in the impossible differential. Discard these 64-bit values from the list of all the 2^{64} possible values of the subkey (k_4, k_6, k_7, k_8).
6. For each guess of (k_1, k_2, k_3), there are several 64-bit key words (k_4, k_6, k_7, k_8) kept after the 2^{63+n}-pair filters. Search for the remaining 16-bit key word k_5, and get the right key. Otherwise, return to Step 2, and repeat the above process.

Complexity Evaluation. In Step 6, the number of remaining values in the list is about $\epsilon = 2^{112}(1 - \frac{1}{2^{64}})^{2^{n+63}}$. To find a balance between the complexity of searching the right key in Step 6 and the complexity of Steps 1-5, we choose $n = 4.5$, and then $\epsilon = 2^{96}$. Then the attack needs about $2^{n+48} = 2^{52.5}$ chosen plaintexts.

In the first step, we need about $2^{n+48} = 2^{52.5}$ encryptions to get the corresponding ciphertexts. Step 2 costs $2^{n+63+32} = 2^{99.5}$ memory accesses to compute k_4. In Steps 3, 4 and 5, we need to access a table of size 2^{48} for each plaintext-ciphertext pair and subkey (k_1, k_2, k_3). Step 6 requires about $2^{96} \times 2^{16} = 2^{112}$ encryptions to search for the correct key. We assume the complexity of one memory access to T_1 and T_2 to be about a one-round encryption. The total complexity of our attack is hence about $2^{n+63} \times 2^{48} \times 3/7 + 2^{112} = 2^{114.3}$ 7-round encryptions.

5 Impossible Differential Attack on the First 7 Rounds of KASUMI

To analyze the first 7 rounds of KASUMI, we specify the 5-round impossible differential as:

$$(0, a_l \| a_r) \overset{5R}{\nrightarrow} (0, a_l \| a_r).$$

Combined with the key words distribution, we mount the 5-round impossible differential from round 2 to round 6, and extend one round forward and backward respectively (see Fig. 3). The difference $a_l \| a_r$ that satisfies Observation 4 is used to make the input difference of $FI_{1,2}$ and $FI_{7,2}$ be 0. Then we utilize Observation 1 to decrease the time complexity. In this case, the difference ΔL_1 and ΔR_6 do not depend on the key word k_4 by key schedule algorithm (see Fig. 3). In the following, we demonstrate a known plaintext attack on 7-round KASUMI.

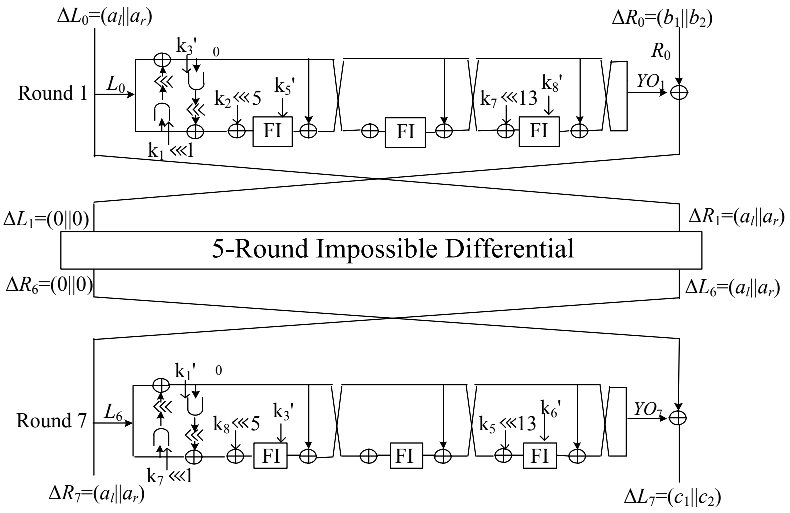


Fig. 3. Impossible differential attack on the first 7 rounds of KASUMI

Data Collection. For 2^m plaintexts $P(L_0, R_0)$, query their ciphertexts $C(L_7, R_7)$, and store the (P, C) pairs in a hash table with index $L_0 \oplus R_7$. There are about 2^{2m-33} pairs whose input and output differences are $(a_l \| a_r, *)$ and $(*, a_l \| a_r)$ respectively, where '*' is any 32-bit value. Save the pairs whose differences $(a_l \| a_r)$ make the equation (5) hold. There are about $2^{2m-33} \times (3/4)^{16} = 2^{2m-39.64}$ pairs kept on average. For the remaining plaintext-ciphertext pairs, the differences satisfy $\Delta L_0 = (a_l \| a_r)$, $\Delta R_7 = (a_l \| a_r)$ and $(a_l \lll 1) \vee \neg a_r = 0xffff$. We use the difference $(a_l \| a_r)$ as the index for kept plaintext-ciphertext pairs.

Key Recovery

1. Guess (k_1, k_7), for all the differences $(a_l \| a_r)$, apply Observation 3 to filter the pairs. Keep the pairs whose differences satisfy equations (3) and (4). By Observation 5, there are $2^{2m-39.64+16-32} = 2^{2m-55.64}$ pairs left for every (k_1, k_7) on average.
2. Guess k_5, for each remaining pair, the input and output differences of $FI_{1,1}$ are computed as $\Delta XI_{1,1} = \Delta L_{0,l}$, $\Delta YI_{1,1} = \Delta R_{0,l}$. Then look up table T_2 to get the input and output values $XI_{1,1}$, $YI_{1,1}$ of $FI_{1,1}$. Compute the input values $XI_{1,3}$ and $XI'_{1,3}$ by partial encryption, and thoutput difference $\Delta YI_{1,3} = \Delta R_{0,l} \oplus \Delta R_{0,r}$. Then k_8 is obtained by accessing table T_1.
3. By partial decryption, the intermediate values $XI_{7,1}$ and $XI'_{7,1}$ are deduced from $L_6 = R_7$ and $L'_6 = R'_7$, and $\Delta YI_{7,1} = \Delta L_{7,l}$. Then access table T_1 to get k_3 and $YI_{7,1}$.
4. $XI_{7,3}$ and $XI'_{7,3}$ are deduced by partial decryption, and the output difference $\Delta YI_{7,3} = \Delta L_{7,l} \oplus \Delta L_{7,r}$, so k_6 is obtained by looking up in table T_1. Then compute YL_1 by the function FL_1, and $k_2 = (YL_{1,l} \oplus XI_{1,1}) \ggg 5$. Thus the key words (k_2, k_3, k_6, k_8) produce the impossible differential, discard it from the list of the 2^{64} possible values and start a new guess.
5. For every guess (k_1, k_5, k_7), there are about $2^{64} \times (1-2^{-64})^{2m-55.64}$ key words (k_2, k_3, k_6, k_8) kept after the $2^{2m-55.64}$ pairs filter. Exhaustively search for the 16-bit key word k_4 for the kept values of key words, and get the right key. Otherwise go to Step 1, and repeat the above process.

Complexity Evaluation. Let $m = 62$. In the data collection process, we need $2^m = 2^{62}$ encryptions and $2^{2m-33} = 2^{91}$ computations of equation (5). There are 3^{16} values for $(a_l \| a_r)$ in total by Ovservation 3. For each (k_1, k_7) and $a_l \| a_r$, compute the equations (3) and (4) in Step 1, which needs $2^{32} \times 3^{16} \times 1/4 \times 1/7$ encryptions of 7-round KASUMI. Steps 2-4 cost $2^{2m-55.64} \times 2^{48} \times 4 = 2^{118.36}$ accesses to memory of size 2^{48} and $2^{2m-55.64} \times 2^{48} = 2^{116.36}$ accesses to memory of size 2^{64}. In Step 5, the expected number of remaining keys is $2^{34.4}$. We spend $2^{48} \times 2^{34.4} \times 2^{16} = 2^{98.4}$ 7-round computations to exhaustively search for the right key. Suppose one memory access is equivalent to one round encryption. Hence the total time complexity is about $2^{98.4} + 2^{118.36} \times 1/7 + 2^{116.36} \times 1/7 = 2^{115.8}$ 7-round encryptions, and the data complexity is about 2^{62} known plaintexts and the memory complexity is estimated by the storage of the pairs, which is $2^{2m-39.64} \times 16 = 2^{84.36}$ bytes.

6 Conclusion

In this paper, we extend the 12-year old impossible differential attack on 6-round KASUMI to 7 rounds, thereby reducing the security margin from 2 rounds to 1 round. We refine the impossible differential by selecting some special input difference values. In order to get the secret key with lower computational complexity we treat FI as a big sbox and construct the difference distribution table with the dependent 16-bit subkey KI. Besides, we give some observations on the FL function with a special input difference, with which we give the first impossible differential attack on the first 7 rounds. The impossible differential attack on the last 7 round needs $2^{114.3}$ encryptions with $2^{52.5}$ chosen plaintexts, and $2^{115.8}$ encryptions with 2^{62} known plaintexts for the first 7 rounds.

Acknowledgments. We would like to thank anonymous reviewers for their very helpful comments on the paper. This work is supported by 973 Program(2013CB834205) and the National Natural Science Foundation of China (61133013 and 60931160442), China Postdoctoral Science Foundation(20110490442) and the European Commission under contract ICT-2007-216646 (ECRYPT II). This author Jiazhe Chen is visiting KU Leuven, ESAT/COSIC (Belgium).

References

1. 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)
2. Biham, E., Dunkelman, O., Keller, N.: A Related-Key Rectangle Attack on the Full KASUMI. In: Roy, B. (ed.) ASIACRYPT 2005. LNCS, vol. 3788, pp. 443–461. Springer, Heidelberg (2005)
3. Blunden, M., Escott, A.: Related Key Attacks on Reduced Round KASUMI. In: Matsui, M. (ed.) FSE 2001. LNCS, vol. 2355, pp. 277–285. Springer, Heidelberg (2002)
4. Dunkelman, O., Keller, N., Shamir, A.: A Practical-Time Related-Key Attack on the KASUMI Cryptosystem Used in GSM and 3G Telephony. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 393–410. Springer, Heidelberg (2010)
5. Kühn, U.: Cryptanalysis of Reduced-Round MISTY. In: Pfitzmann, B. (ed.) EUROCRYPT 2001. LNCS, vol. 2045, pp. 325–339. Springer, Heidelberg (2001)
6. Kühn, U.: Improved Cryptanalysis of MISTY1. In: Daemen, J., Rijmen, V. (eds.) FSE 2002. LNCS, vol. 2365, pp. 61–75. Springer, Heidelberg (2002)
7. Matsui, M.: New Block Encryption Algorithm MISTY. In: Biham, E. (ed.) FSE 1997. LNCS, vol. 1267, pp. 54–68. Springer, Heidelberg (1997)
8. Sugio, N., Aono, H., Hongo, S., Kaneko, T.: A Study on Integral-Interpolation Attack of MISTY1 and KASUMI. In: Computer Security Symposium 2006, pp.173–178 (2006) (in Japanese)
9. Sugio, N., Aono, H., Hongo, S., Kaneko, T.: A Study on Higher Order Differential Attack of KASUMI. IEICE Transactions 90-A(1), 14–21 (2007)

10. Sugio, N., Tanaka, H., Kaneko, T.: A Study on Higher Order Differential Attack of KASUMI. In: 2002 International Symposium on Information Theory and its Applications (2002)
11. 3rd Generation Partnership Project, Technical Specification Group Services and System Aspects, 3G Security, Specification of the 3GPP Confidentiality and Integrity Algorithms; Document 2: KASUMI Specification, V3.1.1 (2001)
12. 3rd Generation Partnership Project, Technical Specification Group Services and System Aspects, 3G Security, Specification of the A5/3 Encryption Algorithms for GSM and ECSD, and the GEA3 Encryption Algorithm for GPRS; Document 4: Design and evaluation report, V6.1.0 (2002)

Meet-in-the-Middle Technique for Integral Attacks against Feistel Ciphers

Yu Sasaki[1] and Lei Wang[2]

[1] NTT Secure Platform Laboratories, NTT Corporation
3-9-11 Midori-cho, Musashino-shi, Tokyo 180-8585 Japan
sasaki.yu@lab.ntt.co.jp
[2] The University of Electro-Communications
1-5-1 Choufugaoka, Choufu-shi, Tokyo, 182-8585 Japan

Abstract. In this paper, an improvement for integral attacks against Feistel ciphers is discussed. The new technique can reduce the complexity of the key recovery phase. This possibly leads to an extension of the number of attacked rounds. In the integral attack, an attacker guesses a part of round keys and performs the partial decryption. The correctness of the guess is judged by examining whether the XOR sum of the results becomes 0 or not. In this paper, it is shown that the computation of the XOR sum of the partial decryptions can be divided into two independent parts if the analysis target adopts the Feistel network or its variant. Then, correct key candidates are efficiently obtained with the meet-in-the-middle approach. The effect of our technique is demonstrated for several Feistel ciphers. Improvements on integral attacks against LBlock, HIGHT, and CLEFIA are presented. Particularly, the number of attacked rounds with integral analysis is extended for LBlock.

Keywords: Integral attack, Meet-in-the-middle, Feistel, Partial-sum, LBlock, HIGHT, CLEFIA.

1 Introduction

The integral attack is a cryptanalytic technique for symmetric-key primitives, which was firstly proposed by Daemen *et al.* to evaluate the security of SQUARE cipher [1], and was later unified as integral attack by Knudsen and Wagner [2]. The crucial part is a construction of an *integral distinguisher*: an attacker prepares a set of plaintexts which contains all possible values for some bytes and has a constant value for the other bytes. All plaintexts in the set are passed to the encryption oracle. Then, the corresponding state after a few rounds has a certain property, *e.g.* the XOR of all texts in the set becomes 0 with probability 1. Throughout the paper, this property is called *balanced*.

A key recovery attack can be constructed by using this property. An attacker appends a few rounds to the end of the distinguisher. After she obtains a set of the ciphertexts, she guesses a part of round keys and performs the partial decryption up to the balanced state. If the guess is correct, the XOR sum of the results always becomes 0. Hence, the key space can be reduced.

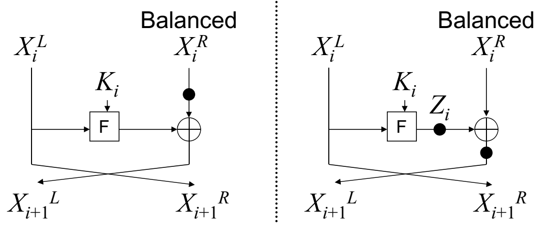


Fig. 1. Left: previous approach, Right: our approach

The application of the integral attack to AES [3, 4] and AES based ciphers is widely known. Moreover, for AES, Ferguson *et al.* proposed an improved technique called *partial-sum* [5], which utilizes the property of an MDS multiplication *e.g.* the MixColumns (MC) operation in AES. It observes that each output byte of the MC operation is a linear combination of four input bytes, (x_0, x_1, x_2, x_3). Therefore the sum of the output value $\bigoplus \mathrm{MC}(x_0, x_1, x_2, x_3)$ can be computed in byte-wise independently, *i.e.* $\bigoplus(a_0 \cdot x_0) \oplus \bigoplus(a_1 \cdot x_1) \oplus \bigoplus(a_2 \cdot x_2) \oplus \bigoplus(a_3 \cdot x_3)$, where a_0, a_1, a_2, a_3 are some coefficients.

Another popular block-cipher construction is the Feistel network, which separates the state X_i to the left half X_i^L and the right half X_i^R. It updates the state by $X_{i+1}^R \leftarrow X_i^L$ and $X_{i+1}^L \leftarrow X_i^R \oplus F(X_i^L, K_i)$, where F is called a round function and K_i is a round key for updating i-th round. Variants of the Feistel network, *e.g.*, generalized or modified Feistel network are also popular designs.

Several papers have already applied the integral attack to ciphers with the Feistel network or its variant. In this paper, we call such ciphers *Feistel ciphers*. Examples are the attacks on Twofish [6], Camellia [7–10], CLEFIA [11, 12], SMS4 [13], Zodiac [14], HIGHT [15], and LBlock [16].

Our Contributions

In this paper, an improvement for integral attacks against Feistel ciphers is discussed. The new technique can reduce the complexity of the key recovery phase. This possibly leads to an extension of the number of attacked rounds. The observation is described in the right-hand side of Fig. 1, which is very simple, but can improve many of previous integral attacks. Assume that the balanced state appears on the state X_i^R, thus an attacker examines if $\bigoplus(X_i^R) = 0$ or not. Due to the linearity of the computation, this can be transformed as $\bigoplus Z_i = \bigoplus X_{i+1}^L$, where Z_i is the state after the round function is applied. Finally, we can compute the left-hand side and right-hand side of this equation independently, and the key candidates that result in the balanced state of X_i^R are identified by checking the matches between two values. The match can be done with the meet-in-the-middle technique [17–19]. Therefore, the efficiency of the attack can be improved.

Table 1. Comparison of attack results

Target	Key size	Approach	#Rounds	Data	Time	Memory (bytes)	Reference
LBlock	80 bits	Imp. Diff.	21	$2^{62.5}$	$2^{73.7}$	$2^{55.5}$	[21]
		RK Imp. Diff.	23	2^{40}	2^{70}	–	[22]
		Integral	18	2^{62}	2^{36}	2^{20}	[16]
		Integral	18	2^{62}	2^{12}	2^{16}	This paper
		Integral	20	$2^{63.6}$	$2^{39.6}$	2^{35}	This paper
HIGHT	128 bits	Imp. Diff.	27	2^{58}	$2^{126.6}$	2^{120}	[23]
		RK Diff.	32	$2^{57.84}$	$2^{123.17}$	–	[24]
		Integral	18	2^{62}	2^{36}	2^{20}	[16]
		Integral	22	2^{62}	$2^{118.71}$	2^{64}	[15]
		Integral	22	2^{62}	$2^{102.35}$	2^{64}	This paper
CLEFIA (-128)	128 bits	Improb. Diff.	13	$2^{126.83}$	$2^{126.83}$	$2^{105.32}$	[25]
		Imp. Diff.	13	$2^{117.8}$	$2^{121.2}$	–	[26]
		Integral	12	$2^{115.7}$	$2^{116.7}$	2^{100}	[11]
		Integral	12	$2^{115.7}$	$2^{103.1}$	$2^{75.2}$	This paper

The complexity for the integral attacks is only for recovering partial key bits and does not include the one for processing the data, while the complexity for the impossible/improbable differential attacks is for the full key recovery.
18-round attacks on LBlock recovers only 16 bits, and the exhaustive search on remaining 64 bits takes 2^{64} computations. However, we can avoid it by iterating the attack for other balanced bytes and recover more key bits.

Moreover, our technique can be combined with the partial-sum technique, which exploits another aspect of the independence in some computation[1].

We demonstrate the effect of our technique by applying it to several Feistel ciphers. The results are summarized in Table 1. The complexities for recovering partial key bits are compared for the integral attacks because this paper mainly focuses on the improvement of the key recovery phase inside the integral attack and does not pay attentions to the trivial additional exhaustive search of remaining key bits. The first application is a block-cipher LBlock [16]. We first show an improvement of the 18-round attack by [16]. [16] claimed that the attack could be extended up to 20 rounds. However, we show that the attack is flawed. Then, we construct a first successful integral attack against 20-round LBlock by using our technique. Moreover, we further reduce the complexity by applying the partial-sum technique. The second application is a block-cipher HIGHT [15]. We first show that the previous 22-round attack can be trivially improved, and then, the complexity is further reduced by using our technique. The last application is a block-cipher CLEFIA [12], which uses the generalized Feistel network, and its round function takes the SP function. We combine the partial-sum technique with our approach, and improve 12-round attack on CLEFIA-128.

[1] The same strategy is used in the dedicated attack on TWINE [20].

Note that for Feistel ciphers, an impossible/improbable differential attack is often the best analysis in the single-key setting. In fact, our approach only works for fewer rounds than those attacks. Nevertheless, we believe that presenting a new approach for improving integral attacks is useful, because integral attacks have already been established as a basic tool of the cryptanalysis. In Table 1, we also list the complexity of the current best related-key attack for comparison.

Paper Outline

The organization of this paper is as follows. Section 2 gives preliminaries. Section 3 explains our basic idea to improve the integral analysis for Feistel ciphers. Section 4 applies our technique to LBlock, HIGHT, and CLEFIA-128. Finally, we conclude this paper in Section 5.

2 Preliminaries

2.1 Notations for Integral Attack

The integral attack is a cryptanalytic technique for symmetric-key primitives, which was firstly proposed by Daemen *et al.* to evaluate the security of the SQUARE cipher [1]. Its brief description has already given in Section 1. To discuss integral distinguishers, the following notations are used in this paper.

"A (Active)" : all values appear exactly the same number in the set of texts.
"B (Balanced)" : the XOR of all texts in the set is 0.
"C (Constant)" : the value is fixed to a constant for all texts in the set.

2.2 Partial-Sum Technique

The partial-sum technique was introduced by Ferguson *et al.* [5] in order to improve the complexity of the key recovery phase in the integral attack. The original attack target was AES. In the key recovery phase of the AES, the partial decryption involves 5 bytes of the key and 4 bytes of the ciphertext. Suppose that the number of data to be analyzed, n, is 2^{32} and the byte position b of each ciphertext is denoted by $C_{b,n}$. Then, the equation can be described as follows.

$$\bigoplus_{n=1}^{2^{32}}\Big[S_4\Big(S_0(c_{0,n}\oplus k_0)\oplus S_1(c_{1,n}\oplus k_1)\oplus S_2(c_{2,n}\oplus k_2)\oplus S_3(c_{3,n}\oplus k_3)\oplus k_4\Big)\Big]. \quad (1)$$

With a straightforward method, the analysis takes $2^{32+40} = 2^{72}$ partial decryptions, while the partial-sum technique can perform this computation only with 2^{48} partial decryptions. The idea is partially computing the sum by guessing each key byte one after another.

The analysis starts from 2^{32} texts $(c_{0,n}, c_{1,n}, c_{2,n}, c_{3,n})$. First, two key bytes k_0 and k_1 are guessed, and $S_0(c_{0,n} \oplus k_0) \oplus S_1(c_{1,n} \oplus k_1)$ is computed for each guess. Let $x_{i,n}$ be $\bigoplus_{p=0}^{i}(S_p(c_{p,n} \oplus k_p))$[2]. Then, $S_0(c_{0,n} \oplus k_0) \oplus S_1(c_{1,n} \oplus k_1)$ can be represented by $x_{1,n}$, and Eq. (1) becomes

$$\bigoplus_{n=1}^{2^{32}} \Big[S_4 \Big(x_{1,n} \oplus S_2(c_{2,n} \oplus k_2) \oplus S_3(c_{3,n} \oplus k_3) \oplus k_4 \Big) \Big].$$

The original set includes 2^{32} texts, but now only 3-byte information (x_1, c_2, c_3) is needed. Hence, by counting how many times each of 3-byte values (x_1, c_2, c_3) appears and by only picking the values that appear odd times, the size of the data set is compressed into 3 bytes. For the second step, a single key byte k_2 is guessed, and the size of the data set becomes 2 bytes (x_2, c_3). For the third step, a single key byte k_3 is guessed, and the size of the data set becomes 1 byte (x_3). Finally, a single byte k_4 is guessed and Eq. (1) is computed for each guess.

The complexity for the guess of k_0, k_1 is $2^{16} \times 2^{32} = 2^{48}$, for the guess of k_2 is $2^{16} \times 2^{8} \times 2^{24} = 2^{48}$. Similarly, the complexity is preserved to be 2^{48} until the last computation.

2.3 Previous Integral Attack for Feistel Ciphers

Assume that the right half of the state in round i, denoted by X_i^R, has the balanced property. To recover the key, many of previous attacks use all information that relates to X_i^R. This is illustrated in the left-hand side of Fig. 1. Let $\#K(X)$ be the number of key bits that need to be guessed to obtain the value of X by the partial decryption. Similarly, let $\#C(X)$ be the number of ciphertext bits that are used to obtain X by the partial decryption. Many of previous attacks spend $2^{\#K(X_i^R)+\#C(X_i^R)}$ computations to obtain $\bigoplus(X_i^R)$.

3 Meet-in-the-Middle Technique for Integral Attacks

In this section, we explain our idea that improves the time complexity and the amount of memory to be used in the key recovery phase. The observation is very simple, but can improve many of previous integral attacks on Feistel ciphers.

Let n be the number of texts. $\bigoplus_n(X_{i,n}^R)$ can be described as $\bigoplus_n(Z_{i,n} \oplus X_{i+1,n}^L)$, where Z_i is the state after the round function is applied. We only use the notation n to show that the sum of the value is later computed. The structure is illustrated in the right-hand side of Fig. 1. Due to the linear computation, the sum of each term can be computed independently. Hence, the equation $\bigoplus_n(X_{i,n}^R) = 0$, can be written as

$$\bigoplus_n Z_{i,n} = \bigoplus_n X_{i+1,n}^L. \tag{2}$$

Then, we compute $\bigoplus_n Z_{i,n}$ for all guesses of $\#K(Z_i)$ and store the result in a table, and independently compute $\bigoplus_n X_{i+1,n}^L$ for all guesses of $\#K(X_{i+1}^L)$ and

[2] Notation $x_{i,n}$ is somehow confusing. $x_{i,n}$ represents the sum of i S-box outputs.

store the result in a table. Finally, the key values that result in the balanced state can be identified with the same manner as the meet-in-the-middle attack *i.e.* by checking the matches between two tables. The time complexity of the attack can be reduced into

$$\max\{2^{\#K(Z_i)+\#C(Z_i)}, 2^{\#K(X_{i+1}^L)+\#C(X_{i+1}^L)}\}. \tag{3}$$

Note that if the key bits to compute Z_i and X_{i+1}^L have some overlap, we can apply the three subset meet-in-the-middle attack [17] *i.e.*, the shared bits are firstly guessed, and for each guess, the other bits are independently computed. Let K_s be a set of bits for the shared key, and $|K_s|$ is the bit number of K_s. Then, the memory complexity of the attack can be reduced into

$$\max\{2^{\#K(Z_i)+\#C(Z_i)-|K_s|}, 2^{\#K(X_{i+1}^L)+\#C(X_{i+1}^L)-|K_s|}\}. \tag{4}$$

Due to the structure of the Feistel network, the first item is always bigger than the second item. Thus, the time and memory complexity is simply written as

$$(\text{Time}, \text{Memory}) = \left(2^{\#K(Z_i)+\#C(Z_i)}, 2^{\#K(Z_i)+\#C(Z_i)-|K_s|}\right). \tag{5}$$

4 Applications of Our Technique

In this section, we demonstrate several applications of our technique by improving previous integral attacks against several ciphers. The goal of this section is to show that our technique can be applied to a wide range of Feistel ciphers. Therefore, we do not optimize each attack by looking inside of the key schedule. We omit its description, and assume that each round key is independent.

4.1 LBlock

LBlock is a light-weight block-cipher proposed at ACNS 2011 by Wu and Zhang [16]. The block size is 64-bits and the key size is 80 bits. It adopts a modified Feistel structure with 32 rounds, and its round function consists of the key addition, an S-box layer, and a permutation of the byte positions. The plaintext is loaded into an internal state $X_0^L \| X_0^R$. The state $X_i^L \| X_i^R$ is updated by using a round key K_i and the round function described in Fig. 2. We denote the j-th byte (1 byte is 4 bits for LBlock) of a 32-bit word X by $X[j]$, where 0-th byte is the right most byte in the figure.

Previous 18-Round Attack. The designers showed a 15-round integral distinguisher. For a set of 2^{60} plaintexts with the form of $(AAAC\ AAAA\ AAAA\ AAAA)$, the state after 15 rounds, $(X_{15}^L \| X_{15}^R)$, has the form of $(????\ ????\ ?B?B\ ?B?B)$. By using this property, the designers showed an 18-round key recovery attack. The key recovery phase is illustrated in Fig. 3. The attacker guesses a part of round keys, and decrypts the ciphertexts up to the fourth byte of X_{15}^R and checks if its sum is 0 or not. As shown in Fig. 3, five bytes of the ciphertext ($X_{18}^L[0,6]$ and $X_{18}^R[1,4,6]$)

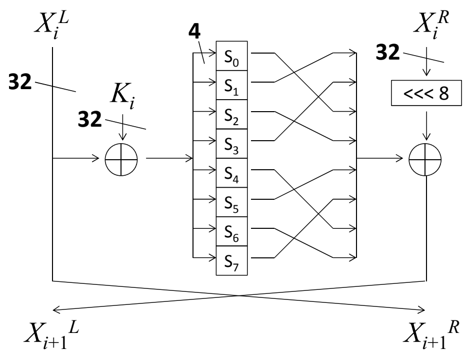


Fig. 2. LBlock Round function

and four bytes of keys ($K_{17}[1,4], K_{16}[4]$, and $K_{15}[4]$) relate to the partial decryption for $X_{15}^R[4]$. The attacker first counts how many times each of 5-byte values $X_{18}^L[0,6], X_{18}^R[1,4,6]$ appears and only picks values that appear odd times. Hence, at most $2^{4*5} = 2^{20}$ values are stored in a memory. Then, for each guess of four key bytes, she computes the corresponding $X_{15}^R[4]$ and computes the sum. The attack complexity is $2^{20} \times 2^{16} = 2^{36}$ partial decryptions.

Improved 18-Round Attack. The attack complexity can be improved by applying our technique. The condition $\bigoplus X_{15}^R[4] = 0$ can be written as

$$\bigoplus Z_{15}[6] = \bigoplus X_{16}^L[6]. \tag{6}$$

As shown in Fig. 3, the computation of $Z_{15}[6]$ involves three bytes of the ciphertext ($X_{18}^L[6]$ and $X_{18}^R[4,6]$) and three bytes of round keys ($K_{17}[4], K_{16}[4]$, and $K_{15}[4]$), which are denoted by numbers in red square brackets. Similarly, the computation of $X_{16}^L[6]$ involves two bytes of the ciphertext ($X_{18}^L[0]$ and $X_{18}^R[1]$) and a single byte of a round key ($K_{17}[1]$), which are denoted by numbers in blue round brackets. The attack procedure is as follows.

1. Query 2^{60} plaintexts which has the form of ($AAAC\ AAAA\ AAAA\ AAAA$).
2. Prepare the memory which stores how many times each three-byte value $X_{18}^L[6], X_{18}^R[4,6]$ appears, and pick the values which appear odd times. Do the same for two-byte values $X_{18}^L[0], X_{18}^R[1]$.
3. For all three-byte keys $K_{17}[4], K_{16}[4]$, and $K_{15}[4]$, compute $Z_{15}[6]$ for all three-byte values $X_{18}^L[6], X_{18}^R[4,6]$ and store its sum in a list $L_{Z_{15}}$.
4. For all values of a single-byte key $K_{17}[1]$, compute $X_{16}^L[6]$ for all two-byte values $X_{18}^L[0], X_{18}^R[1]$ stored in the memory and store its sum in a list $L_{X_{16}^L}$.
5. Check the matches between $L_{Z_{15}}$ and $L_{X_{16}^L}$. If the matches are found, output the corresponding 4-byte keys as correct key candidates.

Step 2 requires a memory to store 2^{12} 3-byte values. Step 3 requires a complexity of $2^{12} \times 2^{12} = 2^{24}$ partial decryptions and a memory to store 2^{12} sums. Step 4

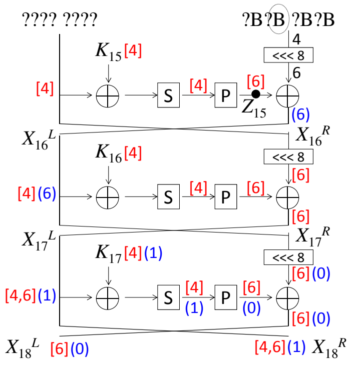


Fig. 3. 18-round attack on LBlock

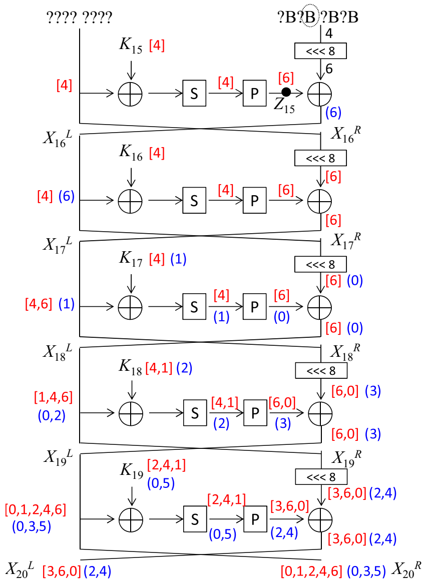


Fig. 4. 20-round attack on LBlock

requires a complexity of $2^4 \times 2^8 = 2^{12}$ partial decryptions and a memory to store 2^4 sums. Step 5 is performed with 2^{12} table look-ups, and $2^{16} \times 2^{-4} = 2^{12}$ values are output as correct key candidates.

By iterating the above steps 3 times, a single key candidate is obtained, which requires 2^{24} 18-round LBlock computations and the memory to store 2^{12} LBlock state. This is faster than the previous 18-round attack. The data complexity is the same as the previous attack, which is $4 \times 2^{60} = 2^{62}$ chosen plaintexts.

Further Improvement with the Partial-Sum Technique. The attack complexity can be further improved with the partial-sum technique. the computation of $\bigoplus Z_{15}[6]$ can be written as follows:

$$\bigoplus S\Big[S\Big(S(X^R_{18}[4] \oplus K_{17}[4]) \oplus X^L_{18}[6] \oplus K_{16}[4]\Big) \oplus X^R_{18}[6] \oplus K_{15}[4]\Big]. \quad (7)$$

In the previous section, this was computed with 2^{24} computations, but we can compute it only with 2^{16} computations with the partial-sum technique. The analysis starts from 3-byte tuple $(X^L_{18}[6], X^R_{18}[4], X^R_{18}[6])$ with 2^{12} data. Firstly a single key byte $K_{17}[4]$ is guessed, and $S(X^R_{18}[4] \oplus K_{17}[4]) \oplus X^L_{18}[6]$ is computed for each guess. Let y_1 be the result. Then, the data can be compressed into 2-byte tuple $(y_1, X^R_{18}[6])$. Secondly, $K_{16}[4]$ is guessed and $S(y_1 \oplus K_{16}[4]) \oplus X^R_{18}[6]$ is computed for each guess. Let y_2 be the result. Then, the data can be compressed into 1-byte y_2. Finally, for each guess of $K_{15}[4]$, $\bigoplus Z_{15}[6]$ is computed by $S(y_2 \oplus K_{15}[4])$.

The guess of $K_{17}[4]$ requires $2^4 \cdot 2^{12} = 2^{16}$ computations. The guess of $K_{16}[4]$ requires $2^4 \cdot 2^4 \cdot 2^8 = 2^{16}$ computations. Finally, the guess of $K_{15}[4]$ requires $2^4 \cdot 2^4 \cdot 2^4 \cdot 2^4 = 2^{16}$ computations. In summary, the time complexity of the attack is reduced into 2^{16}.

Remarks. The 18-round attack only recovers 16 bits of subkeys. Hence, the exhaustive search on the remaining 64 bits costs 2^{64}, which is much more expensive. This can be avoided by performing the above procedure on the other balanced bytes. We stress that the queried data can be shared among the analysis for different balanced bytes. Hence, the data complexity keeps unchanged and only the time and memory complexity increases linearly.

Extension to 20-Round Attack. Wu and Zhang [16] claimed that the attack could be extended up to 20 rounds because only 12 key bytes relate to the partial decryption for $X^R_{15}[4]$. However, we show that this attack is flawed. It is true that 12 key bytes relate to $X^R_{15}[4]$, *i.e.*, $\#K(X^R_{15}[4]) = 48$. However, they did not consider the increase of the number of bytes in the ciphertext that relate to $X^R_{15}[4]$. The analysis is given in Fig. 4. It shows that $\#C(X^R_{15}[4])$ is 48 (12 bytes). Hence, their attack requires $2^{48+48} = 2^{96}$ partial decryptions, which is more expensive than the brute force attack.

In our approach, as shown in Fig. 4, both of $\#K(Z_{15}[6])$ and $\#C(Z_{15}[6])$ are 32 (8 bytes). Hence, the complexity to analyze a single set is reduced to $2^{32+32} = 2^{64}$, which is faster than the brute force attack. For each analysis, the key space becomes 2^{-4}. Hence, by repeating the analysis 12 times, 12 bytes of the key space is reduced to 1. Moreover, similarly to the 18-round attack, the partial-sum technique can be applied to compute $\bigoplus Z_{15}[6]$. The details are omitted due to the limited space. For each guess of a single-key byte, the data is compressed by 1 byte. Hence, the final complexity becomes $2^4 \cdot 2^{32} = 2^{36}$.

Note that, the attack outputs 11 bytes of the key candidates (2^{44}) as a result of the first analysis. To store these candidates, more memory than for storing $\#C(Z_{15}[6])$ is necessary. This can be avoided by analyzing 4 sets of plaintexts simultaneously, and thus the effect of the matching part with the meet-in-the-middle approach becomes 4 times. As a result, the key space after the result of the first analysis becomes 8 bytes, which is the same size as $\#C(Z_{15}[6])$.

In summary, our attack is the first successful integral attack against 20-round LBlock with approximately $12 * 2^{36} \approx 2^{39.6}$ LBlock computations, $8 * 2^{32}$ bytes of memory, and $12 * 2^{60} \approx 2^{63.6}$ chosen plaintexts. The previous attack evaluated that 13 sets of plaintexts are necessary to recover the key with a high success probability. Under the same philosophy, our attack also requires $13 * 2^{36} \approx 2^{39.7}$ LBlock computations, and $2^{63.7}$ chosen plaintexts.

Table 2. Key schedule for the keys that relate to the key recovery phase

SK_{69}^0	SK_{73}	SK_{76}^0	SK_{77}	SK_{80}	SK_{81}	SK_{84}	SK_{85}	SK_{87}^0	WK_5	WK_6	WK_7
MK_1^0	MK_{13}	MK_8^0	MK_9	MK_3	MK_4	MK_7	MK_0	MK_2^0	MK_1	MK_2	MK_3

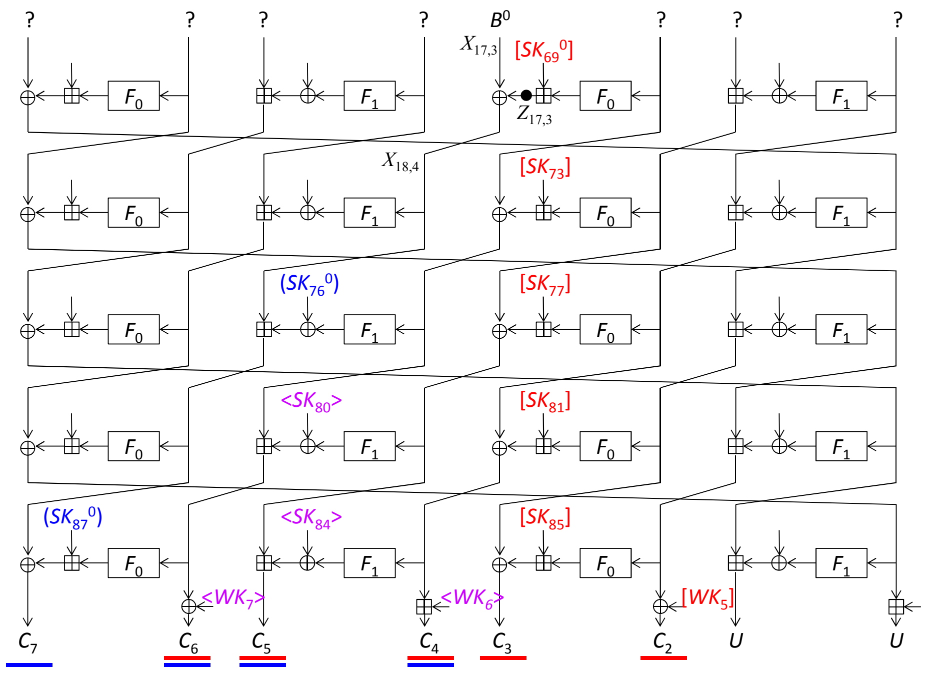


Fig. 5. Key recovery phase for 22-round HIGHT

4.2 HIGHT

HIGHT is a light-weight block-cipher proposed at CHES 2006 by Hong *et al.* [27]. The block size is 64 bits and the key size is 128 bits. It adopts the generalized Feistel structure with 8 branches and 32 rounds. The round function consists of the ARX structure. The plaintext is loaded into an internal state $X_{0,7}\|X_{0,6}\|\cdots\|X_{0,0}$. The state $X_{i,7}\|X_{i,6}\|\cdots\|X_{i,0}$ is updated by using round keys $SK_{4i}, SK_{4i+1}, SK_{4i+2}, SK_{4i+3}$ and the round function. We denote the k-th bit of a byte $X_{i,j}$ by $X_{i,j}^k$. Note that each round key is a copy of a part of the original secret key K, and which part is used is defined in the specification. Because the previous attack by Zheng *et al.* [15] has already exploited the relation of subkeys, we need to consider it to improve the attack.

Previous 22-Round Attack. Zheng *et al.* showed a 17-round integral distinguisher. For a set of 2^{56} plaintexts with the form of (A, A, A, A, A, A, A, C), the state after 17 rounds, $(X_{17,7}\|X_{17,6}\|\cdots\|X_{17,0})$, has the form of $(?, ?, ?, ?, B^0, ?, ?, ?)$, where B^0 stands for the balanced state with respect to the

0-th bit. By using this property, Zheng *et al.* showed a 22-round key recovery attack. The key recovery phase for the 22-round attack is illustrated in Fig. 5.

The attacker prepares a set of 2^{56} plaintexts that satisfies the above form. As shown in Fig. 5, 48 bits of the ciphertext $(C_7, C_6, C_5, C_4, C_3, C_2)$ and 75 bits of round keys $(SK_{69}^0, SK_{73}, SK_{76}^0, SK_{77}, SK_{80}, SK_{81}, SK_{84}, SK_{85}, SK_{87}^0, WK_5, WK_6, WK_7)$ relate to the partial decryption for $X_{17,3}^0$. The related 75 bits of the round keys have some overlap with respect to the original secret key. The key schedule for these keys is given in Table 2. Here, MK stands for "Master Key", which is the original secret key.

By considering Table 2, the number of bits that need to be guessed is 65. Zheng *et al.* guessed all 65 key bits, and applied the partial decryption for all ciphertexts. Therefore, they concluded that the attack complexity to analyze one set was $2^{56} \times 2^{65} = 2^{121}$ partial decryptions. As a result of analyzing one set, the key space can be reduced by 1 bit. Hence, the attack is repeated for 65 sets. Zheng *et al.* showed that the complexity for 65 iterations was $2^{56}(2^{65} + 2^{64} + \cdots + 2^1) \approx 2^{122}$ partial decryptions, which is equivalent to $2^{118.71}$ 22-round HIGHT encryptions.

Simple Improvement of the Previous Attack. We show that the attack by Zheng *et al.* can be improved very simply. Because the partial decryption involves only 48 bits of the ciphertext, the data to be analyzed can be reduced into 2^{48} ciphertexts. This is done by counting how many times each of 48-bit values appears, and only picking values which appear odd times.

Moreover, for the 7th byte of the ciphertext, C_7, only the least significant bit, C_7^0, is needed to compute $X_{17,3}^0$. Therefore, the data to be analyzed can be further reduced into 2^{41} ciphertexts.

In summary, the attack complexity becomes $2^{41}(2^{65} + 2^{64} + \cdots + 2^1) \approx 2^{107}$ partial decryptions, which is equivalent to $2^{103.71}$ 22-round HIGHT encryptions.

Application of Our Technique. By using our technique, the complexity can be further improved. The condition for $\bigoplus X_{17,3}^0 = 0$ is written as $\bigoplus X_{18,4}^0 = \bigoplus Z_{17,3}^0$. The partial decryption for $Z_{17,3}^0$ involves 73 bits of round keys and 40 bits of ciphertexts. The partial decryption for $X_{18,4}^0$ involves 34 bits of round keys and 25 bits of ciphertexts. If the key schedule is considered, $\#K(Z_{17,3}^0)$ is 64 and $\#K(X_{18,4}^0)$ is 26 respectively, while 24 key bits are overlapped. The dominant complexity is the computations for $\bigoplus Z_{17,3}^0$. According to Eq. (5), the time complexity is $2^{40+64} = 2^{104}$ partial decryptions to analyze a single set.

To reduce the key space into 1, the attack is iterated 65 times. However, the attack complexity cannot be evaluated as $2^{40}(2^{64} + 2^{63} + \cdots + 2^1)$ with the meet-in-the-middle approach. This is because the discarded key candidates are uniformly distributed in the key space. Hence, our strategy is applying the meet-in-the-middle approach 2 times to reduce the key space from 2^{65} to 2^{63}, and then perform the previous attack method to further reduce the key space into 1. Finally, the attack complexity is $2^{104} + 2^{104} + 2^{41} \cdot (2^{63} + 2^{62} + \cdots + 2^1) \approx 2^{106}$ partial decryptions.

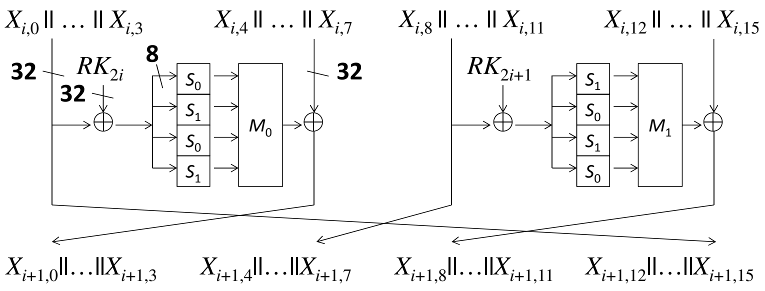

$X_{i,0} \| \dots \| X_{i,3}$
$X_{i,4} \| \dots \| X_{i,7}$
$X_{i,8} \| \dots \| X_{i,11}$
$X_{i,12} \| \dots \| X_{i,15}$
32
RK_{2i}
32
8
32
RK_{2i+1}
S_0
S_1
M_0
M_1
$X_{i+1,0} \| \dots \| X_{i+1,3}$
$X_{i+1,4} \| \dots \| X_{i+1,7}$
$X_{i+1,8} \| \dots \| X_{i+1,11}$
$X_{i+1,12} \| \dots \| X_{i+1,15}$

Fig. 6. Round function of CLEFIA

The previous work compared the complexity for one partial decryption and one HIGHT encryption by counting the number of F functions to be calculated. 7 F_0/F_1 functions are involved in the partial decryption and $22 * 4 = 88$ F_0/F_1 functions are involved in the 22-round HIGHT encryptions. Hence, 2^{106} partial decryptions are equivalent to $2^{106} \times 7/88 \approx 2^{102.35}$ 22-round HIGHT encryptions. Note that 2^{64} key candidates are output as a result of the first analysis. To store these values, the memory to store 2^{64} keys is necessary.

In summary, the attack complexity becomes $2^{102.35}$ 22-round HIGHT encryptions, the memory to store 2^{64} keys, and $65 * 2^{56} \approx 2^{62}$ chosen plaintexts.

4.3 CLEFIA

CLEFIA is a block-cipher proposed at FSE 2007 by Shirai *et al.* [12]. The block size is 128 bits and the key size can be chosen from 128 bits, 192 bits, or 256 bits. It adopts the generalized Feistel structure with 4 branches and 18 rounds for a 128-bit key. The round function consists of the key addition, S-box application, and multiplication by an MDS matrix. The plaintext is loaded into an internal state $X_{0,0} \| X_{0,1} \| \cdots \| X_{0,15}$. The state $X_{i,0} \| X_{i,1} \| \cdots \| X_{i,15}$ is updated by using round keys RK_{2i}, RK_{2i+1} and the round function described in Fig. 6.

Li *et al.* showed a 9-round integral distinguisher for CLEFIA [11]. A set of 2^{112} plaintexts should have the form of $(AAAA\ AAAA\ A'_0 A'_1 A'_2 A'_3\ AAAA)$, where A'_0 is $v \oplus w$, A'_1 is $2v \oplus 8w$, A'_2 is $4v \oplus 2w$, A'_3 is $6v \oplus aw$, and v and w are two active bytes. Then, the state after 9 rounds, $(X_{9,0} \| X_{9,1} \| \cdots \| X_{9,15})$, has the form of $(????\ BBBB\ ????\ ????)$. By using this property, Li *et al.* showed a 11-round basic attack and a 12-round extended attack.

Previous 11-Round Attack. The key recovery phase is described in Fig. 7. An equivalent transformation is applied to the 10th round. $M_0^{-1}(X_{9,4}, X_{9,5}, X_{9,6}, X_{9,7})$ is still a balanced state because M_0 is a linear operation (an MDS matrix multiplication). The attacker guesses round keys and aims to detect if the sum of the 0-th byte of $M_0^{-1}(X_{9,4}, X_{9,5}, X_{9,6}, X_{9,7})$ is 0 or not. The equation that the attacker computes can be written as follows:

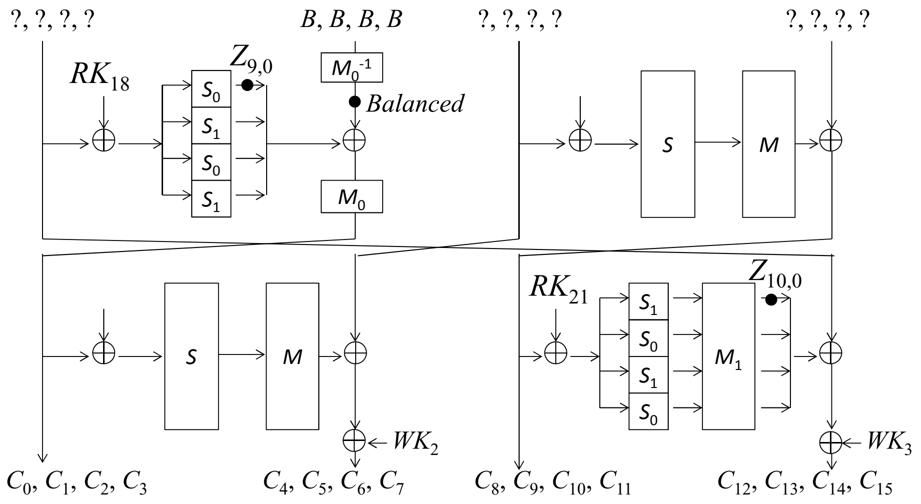


Fig. 7. Key recovery phase for 11-round CLEFIA

$$\begin{aligned}&\bigoplus\Big[S_0\Big(S_1(C_8 \oplus RK_{21,0}) \oplus 08 \cdot S_0(C_9 \oplus RK_{21,1})\oplus \\ &\qquad 02 \cdot S_1(C_{10} \oplus RK_{21,2}) \oplus 0a \cdot S_0(C_{11} \oplus RK_{21,3}) \oplus C_{12} \oplus RK'_{18,0}\Big)\Big] \\ &= \bigoplus C', \end{aligned} \tag{8}$$

where $RK'_{18,0} = WK_{3,0} \oplus RK_{18,0}$ and C' is the 0-th byte of $M_0^{-1}(C_0, C_1, C_2, C_3)$, i.e., $C' = C_0 \oplus 02 \cdot C_1 \oplus 04 \cdot C_2 \oplus 06 \cdot C_3$. The simple method requires $2^{40+40} = 2^{80}$ partial decryptions, while Li *et al.* compute it only with 2^{56} partial decryptions by using the partial-sum technique.

For 2^{112} chosen-plaintexts, the attacker only picks 5-byte values $(C_8, C_9, C_{10}, C_{11}, C_{12})$ and 4-byte values (C_0, C_1, C_2, C_3) that appear odd times. Then, the right-hand side of Eq. (8) can be computed with at most 2^{32} M_0^{-1} computations. The computation for the left-hand side of Eq. (8) starts from 2^{40} texts of 5-byte values $(C_8, C_9, C_{10}, C_{11}, C_{12})$. For simplicity, let $t_0, t_1, \ldots, t_l$ and $r_0, r_1, \ldots, r_l$ be the ciphertext bytes and the corresponding key bytes. Then, let x_i be $\bigoplus_{p=0}^{i} S(t_p \oplus r_p)$. Firstly, the attacker guesses two key bytes r_0 and r_1 and computes x_1. Then, the size of the data to be analyzed can be reduced to 4 bytes $(x_1, C_{10}, C_{11}, C_{12})$. Secondly, the attacker guesses a single key byte r_2, and computes x_2. Then, the size of the data can be reduced to 3 bytes (x_2, C_{11}, C_{12}). Similarly, the attacker guesses r_3 and picks 2-byte data (x_3, C_{12}), then guesses r_4 and computes the final sum. The complexity for computing x_1 is $2^{16} \cdot 2^{40} = 2^{56}$, for computing x_2 is $2^{16} \cdot 2^{8} \cdot 2^{32} = 2^{56}$, and similarly the complexity of 2^{56} is preserved until the final sum is obtained. With the analysis for one data set, the key space becomes 2^{-8} times. Li *et al.* analyzed 6 data sets to uniquely determine the key. The final complexity was estimated as 2^{54} 11-round CLEFIA encryptions.

Improving the Previous 11-Round Attack. We show that the partial-sum technique in the 11-round attack by Li *et al.* can be improved without using our meet-in-the-middle technique. To compute the left-hand side of Eq. (8), Li *et al.* guessed two key bytes r_0 and r_1 to obtain x_1. This procedure seems to come from the original partial-sum application by Ferguson *et al.* [5], which guessed two key bytes at the first step. However, in the analysis for Feistel ciphers, the equation to compute the sum (the left-hand side of Eq. (8) for CLEFIA) has already included a term that only consists of a ciphertext byte (without a key byte). This is actually different from Eq. (1) for AES. Therefore, guessing only a single byte r_0 is enough to compress the data from 2^{40} to 2^{32}.

In details, we firstly guess a single byte $RK_{21,0}$ and compute $S_1(C_8 \oplus RK_{21,0}) \oplus C_{12}$ for 2^{40} texts. Let x'_0 be the result of this computation. We then focus on a 4-byte tuple $x'_0, C_9, C_{10}, C_{11}$, and compress the data size to 2^{32} by only picking the values that appear odd times. For the second step, we guess a single byte $RK_{21,1}$ and compute $08 \cdot S_0(C_9 \oplus RK_{21,1}) \oplus x'_0$ for 2^{32} texts. Let x'_1 be the result. Then, the data size can be reduced to 2^{24} by focusing on 3-byte tuple x'_1, C_{10}, C_{11}. We continue the similar procedure until the final sum is obtained. The complexity for computing x'_0 is $2^8 \cdot 2^{40} = 2^{48}$, for computing x'_1 is $2^8 \cdot 2^8 \cdot 2^{32} = 2^{48}$, and similarly the complexity of 2^{48} is preserved until the final sum is obtained.

In summary, the attack complexity can be reduced by a factor of 2^8, and the total complexity is reduced to approximately 2^{46} 11-round CLEFIA encryptions.

Previous 12-Round Attack. Based on our understandings, we explain the 12-round attack by Li *et al.* The key recovery phase is described in Fig. 8. Several equivalent transformations are applied. An important property is that the whitening key WK_3 only affects the balanced state linearly. Therefore, after taking the sum of 2^{112} texts, the impact of WK_3 disappears. Hereafter, WK_3 is ignored. The equation that the attacker computes can be written as follows:

$$\begin{aligned}\bigoplus\Big[&S_0\Big(S_1(b_0 \oplus RK'_{21,0}) \oplus 08 \cdot S_0(b_1 \oplus RK'_{21,1}) \oplus 02 \cdot S_1(b_2 \oplus RK'_{21,2}) \oplus \\ &\quad 0a \cdot S_0(b_3 \oplus RK'_{21,3}) \oplus C_8 \oplus RK_{18,0}\Big) \oplus \Big(y_1 \cdot S_1(C_8 \oplus RK_{23,0}) \oplus \\ &\quad y_2 \cdot S_0(C_9 \oplus RK_{23,1}) \oplus y_3 \cdot S_1(C_{10} \oplus RK_{23,2}) \oplus y_4 \cdot S_0(C_{11} \oplus RK_{23,3})\Big)\Big] \\ = \bigoplus C',& \end{aligned} \tag{9}$$

where (b_0, b_1, b_2, b_3) is a 4-byte word for $(X_{10,8}, X_{10,9}, X_{10,10}, X_{10,11})$, y_0, y_1, y_2, y_3 are coefficients derived from M_1 and M_0^{-1}, C' is the 0-th byte of $M_0^{-1}(C_{12}, C_{13}, C_{14}, C_{15})$, and RK'_{21} is $RK_{21} \oplus WK_2$. The computation of $\bigoplus C'$ is done with at most 2^{32} computations. The attacker firstly guesses 4 key bytes $RK_{22,0}, \ldots, RK_{22,3}$ and computes (b_0, b_1, b_2, b_3) for all texts. Then, the left-hand side of Eq. (9) is computed with the partial-sum technique. We omit the detailed procedure. Li *et al.* concluded that the complexity to analyze one set is 2^{120} partial decryptions.

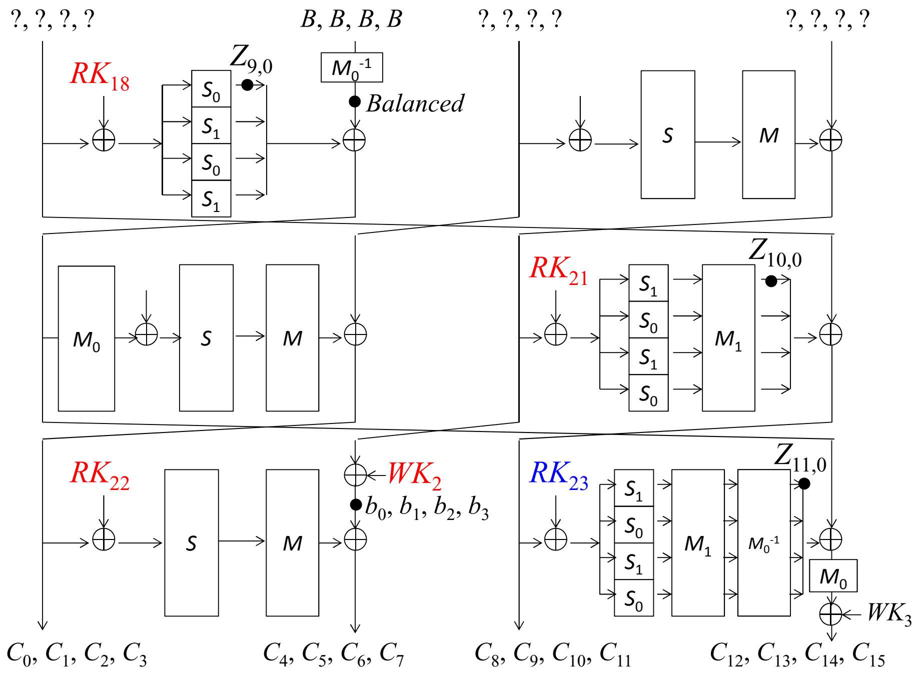


Fig. 8. Key recovery phase for 12-round CLEFIA

Improved 12-Round Attack. The attack by Li *et al.* can be improved by introducing the meet-in-the-middle approach. Eq. (9) is transformed as follows;

$$\bigoplus S_0\Big(S_1(b_0 \oplus RK'_{21,0}) \oplus 08 \cdot S_0(b_1 \oplus RK'_{21,1}) \oplus 02 \cdot S_1(b_2 \oplus RK'_{21,2}) \oplus 0a \cdot S_0(b_3 \oplus RK'_{21,3}) \oplus C_8 \oplus RK_{18,0}\Big)$$
$$= \bigoplus\Big(y_1 \cdot S_1(C_8 \oplus RK_{23,0}) \oplus y_2 \cdot S_0(C_9 \oplus RK_{23,1}) \oplus y_3 \cdot S_1(C_{10} \oplus RK_{23,2}) \oplus y_4 \cdot S_0(C_{11} \oplus RK_{23,3})\Big) \oplus \bigoplus C'. \quad (10)$$

The attack procedure for each set of 2^{112} texts is as follows.

1. With processing 2^{112} plaintexts, we count how many times each value of 9-byte tuple $(C_0, C_1, \ldots, C_8)$, each value of 4-byte tuple $(C_8, C_9, C_{10}, C_{11})$, and each value of 4-byte tuple $(C_{12}, C_{13}, C_{14}, C_{15})$ appears.
2. We compute the second term of the right-hand side of Eq. (10), i.e., $\bigoplus C'$.
3. We compute the first term of the right-hand side of Eq. (10) for the exhaustive guess of RK_{23}, and compute XOR with $\bigoplus C'$. The result, which is the right-hand side of Eq. (10) for each guess of RK_{23}, is stored in a list $L_{X_{10,0}}$.
4. For each guess of RK_{22} (in total 2^{32} iterations), we do as follows.

(a) For 2^{72} texts $(C_0, C_1, \ldots, C_8)$, we compute (b_0, b_1, b_2, b_3) and count how many times each of 5-byte tuple $(b_0, b_1, b_2, b_3, C_8)$ appears. Therefore, the data size is compressed into 2^{40}, and the equation becomes exactly the same as the left-hand side of Eq. (10).

(b) We compute the left-hand side of Eq. (10) by guessing five key bytes $RK'_{21}, RK_{18,0}$ with the same method as our improved 11-round attack. The result is stored in a list $L_{z_{9,0}}$.

5. Finally, we identify right-key candidates by searching for matches between two lists $L_{X_{10,0}}$ and $L_{z_{9,0}}$.

Step 2 requires at most 2^{32} computations. Step 3 requires at most 2^{64} computations with the straightforward method, which is already enough small. This can be further reduced into 2^{48} computations with the partial-sum technique. After Step 3, we obtain a list $L_{X_{10,0}}$ with 2^{32} entries. Step 4(a) requires $2^{32} \cdot 2^{72} = 2^{104}$ partial decryptions. Because our 11-round attack requires 2^{48} partial decryptions, Step 4(b) requires $2^{32} \cdot 2^{48} = 2^{80}$ partial decryptions. As a result of Step 5, we expect to obtain $2^{32+72-8} = 2^{96}$ matches, because the key space is reduced by a factor of 2^8 with the analysis of a single set.

By iterating the analysis with 13 different sets, we expect to obtain a unique solution of 13 key bytes. Note that, by analyzing 4 or more sets simultaneously, the efficiency of the match becomes 4 times or more, and the number of right-key candidates becomes $2^{32+72-(4*8)} = 2^{72}$ or less. This can avoid using 2^{96} memory after the analysis of the first set.

In summary, the bottle-neck of the complexity is Step 4(a), which requires 2^{104} 0.5-round computations. This is equivalent to $2^{104}/24 \approx 2^{99.4}$ 12-round CLEFIA encryptions. After iterating the procedure for 13 sets, the complexity becomes $13 \cdot 2^{99.4} \approx 2^{103.1}$ 12-round CLEFIA computations. The bottle-neck of the memory is for counting how many times each value of 9-byte tuple $(C_0, C_1, \ldots, C_8)$ appears for 2^{112} ciphertexts, which requires 2^{72} 9-byte information. This is equivalent to $2^{75.2}$ bytes or $2^{71.2}$ CLEFIA state. The data complexity is the same as the previous work, which is $13 \cdot 2^{112} \approx 2^{115.7}$ chosen plaintexts.

5 Concluding Remarks

In this paper, we showed an improvement for the integral analysis against Feistel ciphers, which recovers the key by using the meet-in-the-middle approach. We focus on the independence of two computations in the partial decryption for Feistel ciphers, and it reduces the time and memory for the key-recovery phase. Our technique can be combined with the partial-sum technique. We applied our technique for several Feistel ciphers, and showed that the previous integral attacks on LBlock, HIGHT, and CLEFIA-128 could be improved. Particularly, the number of attacked rounds with integral analysis was extended for LBlock.

One possible future work is deriving the limitation of the integral attack, i.e., how many rounds can be potentially attacked by combining currently known techniques such as the meet-in-the-middle and partial-sum techniques. As was done in this paper, the integral attack seems to have more room to be improved.

Then, presenting new techniques is an interesting topic, e.g., application of the partial-sum technique for HIGHT. Because HIGHT adopts two non-commutative operations, XOR and modular addition, the application of the partial-sum is not obvious. We leave it as an open problem.

Acknowledgments. We would like to thank the anonymous reviewers for many helpful comments, especially for the potential application of the partial-sum technique to LBlock.

References

1. 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)
2. Knudsen, L.R., Wagner, D.: Integral Cryptanalysis. In: Daemen, J., Rijmen, V. (eds.) FSE 2002. LNCS, vol. 2365, pp. 112–127. Springer, Heidelberg (2002)
3. Daemen, J., Rijmen, V.: AES Proposal: Rijndael (1998)
4. Daemen, J., Rijmen, V.: The design of Rijndeal: AES – the Advanced Encryption Standard (AES). Springer (2002)
5. Ferguson, N., Kelsey, J., Lucks, S., Schneier, B., Stay, M., Wagner, D., Whiting, D.L.: Improved Cryptanalysis of Rijndael. In: Schneier, B. (ed.) FSE 2000. LNCS, vol. 1978, pp. 213–230. Springer, Heidelberg (2001)
6. Lucks, S.: The Saturation Attack - A Bait for Twofish. In: Matsui, M. (ed.) FSE 2001. LNCS, vol. 2355, pp. 1–15. Springer, Heidelberg (2002)
7. He, Y., Qing, S.: Square Attack on Reduced Camellia Cipher. In: Qing, S., Okamoto, T., Zhou, J. (eds.) ICICS 2001. LNCS, vol. 2229, pp. 238–245. Springer, Heidelberg (2001)
8. Lei, D., Chao, L., Feng, K.: New Observation on Camellia. In: Preneel, B., Tavares, S. (eds.) SAC 2005. LNCS, vol. 3897, pp. 51–64. Springer, Heidelberg (2006)
9. Duo, L., Li, C., Feng, K.: Square Like Attack on Camellia. In: Qing, S., Imai, H., Wang, G. (eds.) ICICS 2007. LNCS, vol. 4861, pp. 269–283. Springer, Heidelberg (2007)
10. Yeom, Y., Park, S., Kim, I.: On the Security of CAMELLIA against the Square Attack. In: Daemen, J., Rijmen, V. (eds.) FSE 2002. LNCS, vol. 2365, pp. 89–99. Springer, Heidelberg (2002)
11. Li, Y., Wu, W., Zhang, L.: Improved Integral Attacks on Reduced-Round CLEFIA Block Cipher. In: Jung, S., Yung, M. (eds.) WISA 2011. LNCS, vol. 7115, pp. 28–39. Springer, Heidelberg (2012)
12. 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)
13. Liu, F., Ji, W., Hu, L., Ding, J., Lv, S., Pyshkin, A., Weinmann, R.-P.: Analysis of the SMS4 Block Cipher. In: Pieprzyk, J., Ghodosi, H., Dawson, E. (eds.) ACISP 2007. LNCS, vol. 4586, pp. 158–170. Springer, Heidelberg (2007)
14. Ji, W., Hu, L.: Square Attack on Reduced-Round Zodiac Cipher. In: Chen, L., Mu, Y., Susilo, W. (eds.) ISPEC 2008. LNCS, vol. 4991, pp. 377–391. Springer, Heidelberg (2008)
15. Zhang, P., Sun, B., Li, C.: Saturation Attack on the Block Cipher HIGHT. In: Garay, J.A., Miyaji, A., Otsuka, A. (eds.) CANS 2009. LNCS, vol. 5888, pp. 76–86. Springer, Heidelberg (2009)

16. Wu, W., Zhang, L.: LBlock: A Lightweight Block Cipher. In: Lopez, J., Tsudik, G. (eds.) ACNS 2011. LNCS, vol. 6715, pp. 327–344. Springer, Heidelberg (2011)
17. Bogdanov, A., Rechberger, C.: A 3-Subset Meet-in-the-Middle Attack: Cryptanalysis of the Lightweight Block Cipher KTANTAN. In: Biryukov, A., Gong, G., Stinson, D.R. (eds.) SAC 2010. LNCS, vol. 6544, pp. 229–240. Springer, Heidelberg (2011)
18. Chaum, D., Evertse, J.-H.: Cryptanalysis of DES with a Reduced Number of Rounds. In: Williams, H.C. (ed.) CRYPTO 1985. LNCS, vol. 218, pp. 192–211. Springer, Heidelberg (1986)
19. Diffie, W., Hellman, M.E.: Exhaustive cryptanalysis of the NBS Data Encryption Standard. Computer 6(10) (1977)
20. Suzaki, T., Minematsu, K., Morioka, S., Kobayashi, E.: TWINE: A Lightweight Block Cipher for Multiple Platforms. In: Knudsen, L.R., Wu, H. (eds.) SAC 2012. LNCS, vol. 7707, pp. 340–355. Springer, Heidelberg (2012)
21. Liu, Y., Gu, D., Liu, Z., Li, W.: Impossible Differential Attacks on Reduced-Round LBlock. In: Ryan, M.D., Smyth, B., Wang, G. (eds.) ISPEC 2012. LNCS, vol. 7232, pp. 97–108. Springer, Heidelberg (2012)
22. Minier, M., Naya-Plasencia, M.: A related key impossible differential attack against 22 rounds of the lightweight block cipher LBlock. Inf. Process. Lett. 112(16), 624–629 (2012)
23. Chen, J., Wang, M., Preneel, B.: Impossible Differential Cryptanalysis of the Lightweight Block Ciphers TEA, XTEA and HIGHT. In: Mitrokotsa, A., Vaudenay, S. (eds.) AFRICACRYPT 2012. LNCS, vol. 7374, pp. 117–137. Springer, Heidelberg (2012)
24. Koo, B., Hong, D., Kwon, D.: Related-Key Attack on the Full HIGHT. In: Rhee, K.-H., Nyang, D. (eds.) ICISC 2010. LNCS, vol. 6829, pp. 49–67. Springer, Heidelberg (2011)
25. Tezcan, C.: The Improbable Differential Attack: Cryptanalysis of Reduced Round CLEFIA. In: Gong, G., Gupta, K.C. (eds.) INDOCRYPT 2010. LNCS, vol. 6498, pp. 197–209. Springer, Heidelberg (2010)
26. Mala, H., Dakhilalian, M., Shakiba, M.: Impossible differential attacks on 13-round CLEFIA-128. J. Comput. Sci. Technol. 26(4), 744–750 (2011)
27. Hong, D., Sung, J., Hong, S.H., Lim, J.-I., Lee, S.-J., Koo, B.-S., Lee, C.-H., Chang, D., Lee, J., Jeong, K., Kim, H., Kim, J.-S., Chee, S.: HIGHT: A New Block Cipher Suitable for Low-Resource Device. In: Goubin, L., Matsui, M. (eds.) CHES 2006. LNCS, vol. 4249, pp. 46–59. Springer, Heidelberg (2006)

Attacking (EC)DSA Given Only an Implicit Hint*

Jean-Charles Faugère[1], Christopher Goyet[1,2], and Guénaël Renault[1]

[1] UPMC, Université Paris 6, LIP6
INRIA, Centre Paris-Rocquencourt, PolSys Project-team
CNRS, UMR 7606, LIP6
4, Place Jussieu
75252 Paris, Cedex 5, France
[2] Thales Communications & Security
160 Boulevard de Valmy
92700 Colombes, France
jean-charles.faugere@inria.fr, christopher.goyet@thalesgroup.com,
guenael.renault@lip6.fr

Abstract. We describe a lattice attack on DSA-like signature schemes under the assumption that implicit information on the ephemeral keys is known. Inspired by the implicit oracle of May and Ritzenhofen presented in the context of RSA (PKC2009), we assume that the ephemeral keys share a certain amount of bits without knowing the value of the shared bits. This work also extends results of Leadbitter, Page and Smart (CHES2004) which use a very similar type of partial information leakage. By eliminating the shared blocks of bits between the ephemeral keys, we provide lattices of small dimension (e.g. equal to the number of signatures) and thus obtain an efficient attack. More precisely, by using the LLL algorithm, the complexity of the attack is polynomial. We show that this method can work when ephemeral keys share certain amount of MSBs and/or LSBs, as well as contiguous blocks of shared bits in the middle. Under the Gaussian heuristic assumption, theoretical bounds on the number of shared bits in function of the number of signed messages are proven. Experimental results show that we are often able to go a few bits beyond the theoretical bound. For instance, if only 2 shared LSBs on each ephemeral keys of 200 signed messages (with no knowledge about the secret key) then the attack reveals the secret key. The success rate of this attack is about 90% when only 1 LSB is shared on each ephemeral keys associated with about 400 signed messages.

Keywords: DLP, ECDSA, Lattice attack, Oracle, Implicit information.

1 Introduction

The security of the main public-key cryptosystems is based on the difficulty of solving certain mathematical problems. In this context, the most commonly used problems come from Number Theory, most notably the integer factorization problem and

* This work was partly supported by the Commission of the European Communities through the ICT program under contract ICT-2007-216676 (ECRYPT-II) and by the French ANR under the Computer Algebra and Cryptography (CAC) project (ANR-09-JCJCJ-0064-01).

the discrete logarithm on finite cyclic groups. For instance, an efficient factorization leads immediately to an attack on the RSA cryptosystem. The security of RSA is then partly based on the presumed difficulty of factoring large integers. Indeed, the most efficient published factoring algorithms have sub exponential asymptotic running times ([Pom84, Len87, LL93]) and it is not known whether efficient factorization can be done in polynomial time on a classical Turing machine. Another classical example is the discrete logarithm problem on a finite cyclic group, upon which is based the security of the ElGamal encryption system, the Diffie-Hellman key exchange and the DSA-like signature schemes. Despite the proven fact that a generic algorithm for computing discrete logarithms in any group is necessarily an exponential algorithm ([Tes01, Sho97, Sha71, Pol78, Pol00]), once again, subexponential algorithms are known to solve some instances of the discrete logarithm problem, i.e. on well-defined classes of groups. For instance, discrete logarithms can be computed in subexponential time on the multiplicative group of finite fields ([AD93]) or on some hyperelliptic curves ([ADH94, ADH99]). Conversely, there is no known subexponential algorithm to solve the discrete logarithm problem on the group of rational points of a well-chosen elliptic curve.

Instead of trying to solve directly a hard mathematical problem, we can rather look at which information should be added in order to solve this problem in polynomial time. With this objective in mind, Rivest and Shamir introduced in [RS86] the notion of oracle to formalize this approach and showed that a RSA modulus $N = pq$ of bit size n (with p and q balanced prime factors of n) can be factored in polynomial time as soon as the $n/3$ most significant bits (MSB) from one of the factors are known. This result was next improved with the so-called Coppersmith's method based on lattice basis reduction as introduced in [Cop96], and which reduced the number of needed bits known to only one-half of the MSB of one of the factors. Beyond the theoretical interest, this additional information can be provided for instance with the help of side-channel analysis and the discovery of some leaks of secret information in some cryptographic systems, and brought to the attacker under the form of an oracle.

In this article, we focus on the Digital Signature Algorithm (DSA) [FIP94] of which the security is based on the difficulty of computing discrete logarithms. The Elliptic Curve Digital Signature Algorithm (ECDSA) [JMV01] is the elliptic curve variant of DSA. The ElGamal [ElG85] and the Schnorr [Sch90] digital signature schemes are also variants of DSA but are rarely used in practice. Anyway, we note that all results presented in this article could be applied to any of these variants. Without redefining the DSA-like schemes (see section 2 for details), we only recall that each user is associated with a pair of private key/public key, such that the private key is the discrete logarithm in a given group of the public key. The user private key and a randomly generated number, called the ephemeral key, are required to compute the signature of a message. The ephemeral key must remain secret and is to be renewed for any new message to be signed.

The first proposal of using an oracle on DSA comes from Howgrave-Graham and Smart in [HGS01] using the LLL lattice reduction algorithm ([LLL82]) to take benefit from the knowledge of a small number of bits in many ephemeral keys. For instance, they show experimentally that if only 8 bits out of 160 bits are known from each ephemeral keys for 30 signed messages, then the secret key is known in less than 10 seconds. However, these results were only heuristics, even though confirmed by experimentation. Nguyen and Shparlinski then presented in [NS02] the first polynomial time algorithm that provably recovers the secret DSA key if about $\log^{1/2}(q)$ LSB (or MSB) of each ephemeral key are known (q denoting the order of the chosen group, see section 2) for a polynomially bounded number of corresponding signed messages. They also show that the case of arbitrary consecutive bits requires much more known bits (about twice as much). Finally, in addition of proving the heuristic attack of [HGS01], Nguyen and Shparlinski ([NS02]) also improved the experimental results of [HGS01] by showing that only 3 known bits of each ephemeral key for 100 signed messages are enough to make the attack feasible. The previous attack is adapted to the case of ECDSA [NS03] and other DSA-like signature schemes like the Nyberg-Rueppel variants of DSA [MNS01]. It is also necessary to mention another analysis that shows the threats associated with the use of private keys generated from an imperfect source of randomness. The attack of Bellare, Goldwasser and Micciancio [BGM97] shows that DSA is totally insecure if private keys are produced by weak pseudo-random number generator such as the Knuth's linear congruential generator. When private keys are smaller than a certain bound, Poulakis proposed in [Pou09] an attack on (EC)DSA using the LLL algorithm and an algorithm to compute the integral points of a class of conics.

Note that unlike the case of RSA, where the oracle gives a way to directly compute the factors of the modulus, all these methods against the DSA-Like cryptosystems bypass the problem of computing discrete logarithm, but rather take advantage of the particular form of the modular equality defining the signature (see (1) in section 2).

At PKC 2009 [MR09], May and Ritzenhofen, in the context of factorization, highly restricted the power of the oracle. They did not assume that the oracle explicitly outputs bits but rather provides only implicit information. This unusual oracle applied against the cryptosystem RSA was formalized as follows: given an RSA modulus $N_1 = p_1q_1$ as input, the oracle outputs a different RSA modulus $N_2 = p_2q_2$ such that the factors of N_2 shared a certain amount of bits with the factors of the modulus N_1. The implicit nature of the information given by the oracle is due to the fact that the value of the shared bits remains unknown as long as the modulus N_1 or N_2 are not factored. Surprisingly, May and Ritzenhofen give an efficient lattice-based algorithm that provably factors N_1 and N_2 in quadratic time provided that factors of the two moduli shared enough of their least significant bits (LSB). They also showed that this algorithm extends to an algorithm with more than one oracle query, which improves upon the required number of shared LSB. This cryptanalysis with the help of an implicit oracle was next extended to the case of shared MSB (and both LSB/MSB) in [FMR10, SM09] and the bound on the required number of shared bits was also improved.

In the case of DSA, an attack using implicit information of a totally different kind was already proposed by Leadbitter, Page and Smart in [LPS04] and made effective in [Tak06a, Tak06b]. From a theoretical point of view, this attack can also be formalized by queries to an oracle which returns a message signed with an ephemeral key of the form $k = y + 2^w y + 2^{2w}x$, i.e. such that the w first bits of k are equal to the w next bits. Experiments show that the repetition of a 4-bits window in the ephemeral keys of 20

Table 1. Summary of our results

Inner product	Canonical		Weighted Euclidean	
Lattice spanned by	the basis M (6) of section 4.1	linearly dependent rows vectors of the matrix (8) obtained by removing the second column of M	the basis M (6) of section 4.1	linearly dependent rows vectors of the matrix (8) obtained by removing the second column of M
constraints on the secret key $\mathbf{a}$	$\mathbf{a} \leq 2^{N-\delta}$ or exhaustive search on the δ msb (section 4.2.1)	No constraint ($\mathbf{a}$ can be up to N bits)	$\mathbf{a} \leq 2^{N-\delta}$ or exhaustive search on the δ msb (section 4.2.1)	No constraint ($\mathbf{a}$ can be up to N bits)
Bounds on the number of shared bits δ function of the number of messages n	$\delta \geq \frac{2N+(n-1)}{n+1} + \frac{c-\log_2(\frac{n+1}{n})}{2}$ (Theorem 1)	$\delta \geq \frac{2N+(n-2)}{n} + \frac{c-\log_2(\frac{n}{n-1})}{2}$ (Theorem 2)	$\delta \geq \frac{N+(n-1)}{n} + \frac{c(n+1)}{2n}$ (Theorem 3a)	$\delta \geq \frac{N+(n-2)}{n-1} + \frac{cn}{2(n-1)}$ (Theorem 3b)

signatures is enough to recover the secret key. This attack is motivated by side-channel analysis. According to the authors, recovering some relation amongst the bits of the secret keys (called "second order leakage") is much more probable than determining the values of such bits because of implementation protections against side-channel analysis. They also described many realistic scenarii where this type of leakage could occur.

Our contribution is to define an attack on DSA-like schemes using implicit information, like in [LPS04], but with an oracle very similar to the one introduced by May and Ritzenhofen ([MR09]). More precisely, we assume that on input of a signed message (m_1, s_1) an oracle outputs different signed message (m_2, s_2) signed with the same secret key and such that the ephemeral keys k_1 and k_2 (used to signed the messages m_1 and m_2 respectively) share a certain amount δ of bits. This oracle only gives implicit information about the bits of the ephemeral keys because the value of the shared bits remains unknown as long as the ephemeral keys stay unknown (or equivalently as long as the secret key stays unknown). In other words, we only know that there are equalities between δ bits of the unknown ephemeral keys used to sign the given messages. We show that this implicit information should be extracted by constructing a lattice which contains a very short vector such that its components yield the secret key. The attack succeeds when this vector is found by the LLL lattice reduction algorithm ([LLL82]), that is when it is small enough. This happens when the ephemeral keys share enough bits δ. This method also works for an arbitrary number n of oracle's queries, each new piece of information decreasing the number of required shared bits.

As usual in lattice basis reduction problems, we have to use the Gaussian heuristic to find a condition on the number of shared bits δ in function of the number of messages n for this vector to be the shortest of the lattice. This condition can be improved by the use of a weighted Euclidean inner product instead of the canonical inner product during the reduction algorithm. A variant of this lattice is also proposed so that the complexity of the attack becomes independent of the secret key size and is polynomial time in n (assuming that the bound on δ is verified). In this case, the lattice is spanned by a set of linearly dependent vectors and the condition on δ is slightly deteriorated. A summary of our method and the proposed improvements can be found in table 1.

As an example of our results, the theorem 3 proves that under the Gaussian heuristic assumption, only 4 LSBs shared on each ephemeral keys of 100 signed messages are enough to make a never-failing attack and that with only 3 LSBs shared, the method needs about 200 signed messages. The result of experiments confirms these theoretical values with a computation time less than 5s, and they even show that the number of messages can be most of the time reduced to an amount comparable to the one which is described in [NS02] despite of the weakness of the oracle. However, these experiments also showed that the success rate of this attack is about 90% when only 1 LSB is shared on each ephemeral key of about 400 signed messages. Interestingly enough, this improves the experimental results of [NS02] where the best experiment corresponds to 3 LSB known (even though LSB are not even known in our contribution).

Throughout this paper, we use common results on euclidean lattice summarized in Appendix A. Section 2 recall the (EC)DSA signature algorithm. Section 3 provides a concrete scenario to justify the existence of this implicit information. In section 4, we present our method by beginning with the case of shared MSB and LSB (subsection 4.1). Essential improvements are proposed in subsection 4.2. In subsection 4.3, we present our method when they are many blocks of shared bits. Finally, we present the result of our experiments in section 5.

2 DSA-Style Signature Scheme

The Digital Signature Algorithm was adopted in 1993 by the U.S. government's National Institute of Standards and Technology (NIST) to become the Digital Signature Standard (DSS) [FIP09]. It is much more used than ElGamal [ElG85] and Schnorr [Sch90] digital signature schemes which are variants of DSA. Thus, we focus on DSA although our attack is transferable to others.

The (EC)DSA algorithm ([FIP09, JMV01]) is defined over a finite abelian group G of prime order q. The group G is chosen as a subgroup of $\mathbb{F}^\times$ (resp. $E(\mathbb{F})$) for DSA (resp. ECDSA) where $\mathbb{F}$ is a finite field (resp. $E(\mathbb{F})$ the group of rational points of an elliptic curve defined over $\mathbb{F}$). For security reason, the size of q is chosen to be at least 160 bits. More precisely, the last revision of the standard ([FIP09, JMV01]) specifies that the parameter q must verify $2^{N-1} < q < 2^N$ where $N \in \{160, 224, 256\}$. The private key is an integer $\mathbf{a} \in \{1, \ldots, q-1\}$ and the public key is the group element $A = g^{\mathbf{a}}$ where g is a publicly known generator of G. Let the function $f : G \longrightarrow \mathbb{F}_q$ be defined by

$$f : \begin{cases} x \in G \subset \mathbb{F}_p^\times & \longmapsto x \bmod q \quad \text{for DSA} \\ (x,y) \in G \subset E(\mathbb{Z}_p) & \longmapsto x \bmod q \quad \text{for ECDSA} \end{cases}$$

The signer chooses an hash function h mapping messages to G. To sign a message m, he chooses a random number $\mathbf{k} \in \{1, \ldots, q-1\}$ called the *ephemeral key* and computes (here we present the computation in the case of DSA only)

$$r = f(g^{\mathbf{k}}) \quad \text{and} \quad s = \mathbf{k}^{-1}(h(m) + \mathbf{a}r) \bmod q \tag{1}$$

The signature on the message m is then the pair (r, s). The verification of the signature is performed by checking

$$f(g^{s^{-1}h(m) \bmod q} \quad A^{s^{-1}r \bmod q}) = f(g^{s^{-1}(h(m)+\mathbf{a}r) \bmod q}) = f(g^{\mathbf{k}}) = r$$

3 Possible Application Scenario

As stated in [LPS04] in reference to side-channel analysis, ”the assumption that an attacker may be able to determine a specific set of bits from the ephemeral secrets is less probable than when the original attacks were first published. It is far more probable that second order, seemingly innocuous information can still be recovered and used by the attacker, even if a defense against the first order leakage is implemented”. Indeed, there are always many situations where implicit information can be found despite the implementation of typically recommended countermeasures. In this paper, as in [LPS04], the attacker is only assumed to be able to determine relations of equality between bits of the ephemeral keys. In addition to the three scenarii given as examples in [LPS04] where implicit information is collected by a power analysis or a timing attack (involving a fixed table implementation of elliptic curve point multiplication, address-bit DPA and cache analysis), we suggest other scenarii which are specifically relevant to our model.

Using invasive attacks, an attacker could lock some bits of the register or memory containing the ephemeral key. This kind of attack largely depends on the implementation and requires a good knowledge of the target, which presupposes at least a partial reverse engineering of the chip. Lasers are then used to cut some wires or to modify the chip (see Skorobogatov thesis [Sko05]). More generally, a lot of fault attacks assume that some bits of the memory are flipped to zero (as in [NNTW05]). But it seems more general to assume that the attacked bits take an indeterminate value.

In addition to weak and wrong implementations (e.g. [GJQ97]), we could also think to destructive applications with a malicious manipulation of random generators for instance. This application could be possible on both embedded systems and software implementation, for instance with physical disturbances, invasive attacks or with malicious softwares. Moreover, the presence of a random number generator testing suite ([Bro11, RSN+10]) does not seem to be an effective countermeasure (see experimental results in section 5.3).

4 Embedding into a Lattice Problem

In this section, we study the security of (EC)DSA given a set of messages signed with the same secret key, and such that the secret ephemeral keys share a certain amount of bits. We recall that the values of these common bits are unknown to us. Thus, the information about the unknown ephemeral keys are implicitly given by the set of signed messages. In other words, we only know that there are some relations amongst the bits of the ephemeral keys used to sign the messages. To ease the exposition of this method, the bit length of the modulus q is noted by N (i.e. we have $2^{N-1} < q < 2^N$).

We will show how the secret key can be revealed by lattice basis reduction provided that there are enough messages or relations between the bits of the ephemeral keys. These constraints are estimated by relating our method to the Gaussian heuristic (see theorem 6 of Appendix A). Since all the complexities given in this paper depend on the complexity of computing a shortest vector in a given lattice, we set the following notation.

Notation 1. *The time complexity of computing a shortest vector of a d-dimensional lattice L of $\mathbb{Z}^n$ will be denoted by $\mathscr{C}(d,B)$, where $B = \log\max_i(\|\mathbf{b_i}\|)$. Notice that computing a shortest vector of any lattice is a NP-hard problem ([Ajt98]) called the Shortest Vector Problem (SVP).*

For certain families of lattices (i.e. under certain conditions on lattices), the shortest vector can be computed with the LLL Algorithm. In this case, we have that $\mathscr{C}(d,B) = \mathscr{O}(d^5(d+B)B)$, i.e. polynomial time in d and B ([NS05], see Appendix A). In this paper, we seek to always stay in this situation.

We first present the case when most significant bits (MSB) and/or less significant bits (LSB) are shared and next the case when blocks of bits are shared.

4.1 Shared MSB and LSB

We first assume that we have n messages m_i $(i = 1,\dots,n)$ with the associated signatures (r_i,s_i) such that all the corresponding ephemeral keys $\mathbf{k}_i$ share a total of δ bits between the MSB and LSB independently of i (see Figure 1). Thus, they are of the form

$$\mathbf{k}_i = \mathbf{k} + 2^t\tilde{\mathbf{k}_i} + 2^{t'}\mathbf{k}' \quad \text{for all } i = 1,\dots,n \tag{2}$$

where

$$0 \le \mathbf{k} < 2^t, \quad 0 \le \mathbf{k}' < 2^{N-t'}, \quad \delta = N - t' + t \quad \text{and} \quad 0 \le \tilde{\mathbf{k}_i} < 2^{N-\delta}$$

with $\mathbf{k}$ and $\mathbf{k}'$ common for all the $\mathbf{k}_i$ (i.e. independent of i).

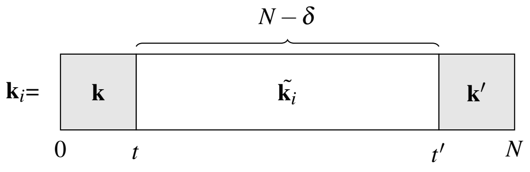


Fig. 1. Ephemeral keys

Note that all the values of $\mathbf{k}_i$, $\mathbf{k}$, $\tilde{\mathbf{k}_i}$ and $\mathbf{k}'$ are unknown. In the n equations (1) defining the signature

$$\begin{cases} m_1 + \mathbf{a}r_1 - s_1\mathbf{k}_1 \equiv 0 & \pmod q \\ m_2 + \mathbf{a}r_2 - s_2\mathbf{k}_2 \equiv 0 & \pmod q \\ \quad\vdots \qquad \vdots \qquad \vdots & \quad\vdots \\ m_n + \mathbf{a}r_n - s_n\mathbf{k}_n \equiv 0 & \pmod q \end{cases} \tag{3}$$

we substitute the $\mathbf{k}_i$ by (2) and eliminate the common variables $\mathbf{k}$ and $\mathbf{k}'$. Then we have

$$\begin{cases} (s_1^{-1}m_1 - s_2^{-1}m_2) + \mathbf{a}(s_1^{-1}r_1 - s_2^{-1}r_2) - 2^t(\tilde{\mathbf{k}_1} - \tilde{\mathbf{k}_2}) \equiv 0 & \pmod q \\ (s_1^{-1}m_1 - s_3^{-1}m_3) + \mathbf{a}(s_1^{-1}r_1 - s_3^{-1}r_3) - 2^t(\tilde{\mathbf{k}_1} - \tilde{\mathbf{k}_3}) \equiv 0 & \pmod q \\ \quad\vdots \qquad\qquad \vdots \qquad\qquad \vdots & \quad\vdots \\ (s_1^{-1}m_1 - s_n^{-1}m_n) + \mathbf{a}(s_1^{-1}r_1 - s_n^{-1}r_n) - 2^t(\tilde{\mathbf{k}_1} - \tilde{\mathbf{k}_\mathbf{n}}) \equiv 0 & \pmod q \end{cases} \tag{4}$$

Let α_i, β_i, $\kappa_i \in \mathbb{Z}$ be such that

$$\begin{cases} \alpha_i := 2^{-t}(s_1^{-1}m_1 - s_i^{-1}m_i) \mod q \\ \beta_i := 2^{-t}(s_1^{-1}r_1 - s_i^{-1}r_i) \quad \mod q \\ \kappa_i := \tilde{\mathbf{k}}_1 - \tilde{\mathbf{k}}_i \end{cases}$$

then (4) becomes

$$\begin{cases} \alpha_2 + \mathbf{a}\beta_2 - \kappa_2 \equiv 0 \pmod q \\ \alpha_3 + \mathbf{a}\beta_3 - \kappa_3 \equiv 0 \pmod q \\ \vdots \quad \vdots \quad \vdots \quad \vdots \\ \alpha_n + \mathbf{a}\beta_n - \kappa_n \equiv 0 \pmod q \end{cases} \tag{5}$$

where $\mathbf{a}$ and κ_i are unknown, β_i and α_i are known. The set of solutions

$$L = \{(x_0, x_1, \dots, x_n) \in \mathbb{Z}^{n+1} | x_0\alpha_i + x_1\beta_i - x_i \equiv 0 \pmod q \text{ for all } i = 2, \dots, n\}$$

forms an $(n+1)$-dimensional lattice spanned by the row vectors of the following basis matrix

$$M = \begin{pmatrix} 1 & 0 & \alpha_2 & \dots & \alpha_n \\ 0 & 1 & \beta_2 & \dots & \beta_n \\ 0 & 0 & q & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & q \end{pmatrix} \tag{6}$$

Note that $v_0 = (1, \mathbf{a}, \kappa_2, \kappa_3, \dots, \kappa_n)$ is an element of the lattice L. Indeed by (5), there are $\lambda_2, \dots, \lambda_n \in \mathbb{Z}$ such that

$$(1, \mathbf{a}, \lambda_2, \dots, \lambda_n) \cdot M = v_0 \tag{7}$$

If we were able to find this vector v_0 in L, then we could recover the secret key $\mathbf{a}$. Thus, we would like to give some conditions so that the vector v_0 be a short vector in L, and therefore may be obtained by lattice basis reduction.

However, it is easy to see that the norm of v_0 is lower bounded by the secret key $\mathbf{a}$, which can be an integer of roughly N bits. The second component of v_0 is then much bigger than the next ones which are $(N-\delta)$-bits integers. Actually, v_0 has no reason to be a short vector of L while $\mathbf{a}$ is so high. Therefore we will first assume that the secret key $\mathbf{a}$ is smaller than $2^{N-\delta}$, before adapting the lattice to be able to find the secret keys up to N-bit size.

This temporary assumption makes v_0 short in the lattice L, but we are still unable to prove that it is the shortest vector of L. Therefore, the Gaussian heuristic (see Appendix A), which is usually applied in this situation, gives us a way to estimate the required number δ of shared bits in function of the number of available messages so that v_0 is likely to be the shortest vector of L.

Assumption 1. *The Gaussian heuristic (theorem 6 of Appendix A) holds with the lattice L. Thus, if v_0 is shorter than the Gaussian heuristic $\lambda_1(L) \approx \sqrt{\frac{d}{2\pi e}}\mathrm{Vol}(L)^{\frac{1}{d}}$ then it is a shortest vector of L.*

Experiments of the section 5 confirm this assumption which seems to be true in practice with the lattices used in our method.

Theorem 1. *Let n messages m_i ($i = 1, \ldots, n$) with the associated signatures (r_i, s_i) such that the ephemeral keys $\mathbf{k}_i$ share a total of δ bits between the MSB and LSB. Under assumption 1 and, under the assumption that the secret key $\mathbf{a}$ is smaller than $2^{N-\delta}$, then $\mathbf{a}$ can be computed in time $\mathscr{C}(n+1, \frac{1}{2}\log_2(n-1)+N)$ as soon as*

$$\delta \geq \frac{2N+(n-1)}{n+1} + \frac{1+\log_2(\pi e) - \log_2(\frac{n+1}{n})}{2}$$

Proof. First of all, we find an upper-bound for the norm of v_0. Under our assumptions, each coefficient of v_0 is an integer of about $(N-\delta)$ bits (except the first one which is equal to 1). Thus, we have the following inequality:

$$\|v_0\|^2 \leq \sum_{i=1}^{n} 2^{2(N-\delta)} = 2^{2(N-\delta)+\log_2 n}$$

On the other hand, thanks to the upper-triangular shape of the matrix M, the volume of L is easily computed as $\mathrm{Vol}(L) = q^{(n-1)} > 2^{(N-1)(n-1)}$. We now seek the condition on δ and n under which the norm of v_0 is smaller than the Gaussian heuristic:

$$2^{2(N-\delta)+\log_2(n)} \leq \frac{n+1}{2\pi e} 2^{2(N-1)\frac{n-1}{n+1}}$$

which is equivalent to

$$\delta \geq \frac{2N+(n-1)}{n+1} + \frac{1+\log_2(\pi e) - \log_2(\frac{n+1}{n})}{2} \qquad \square$$

4.2 Proposal for Improvements

Until now, we made the assumption that $\mathbf{a} \leq 2^{N-\delta}$ to simplify the presentation. Actually, this assumption is rarely verified by the secret key, so we have to adapt the previous attack to be able to find the secret key up to N-bit long. We suggest three ways, which may be concurrent to each other, to reach this goal.

4.2.1 Exhaustive Search

If δ is reasonably small then the method comes down to the previous one with an exhaustive search on the δ-most significant bits of $\mathbf{a}$. Indeed, we have $\mathbf{a} = \tilde{\mathbf{a}} + 2^{N-\delta}a'$ with $\tilde{\mathbf{a}} < 2^{N-\delta}$ and $a' < 2^{\delta}$. An exhaustive search is then made on a' with the previous lattice L in which we set $\alpha_i = 2^{-t}(s_1^{-1}m_1 - s_i^{-1}m_i) + a'2^{-t}(s_1^{-1}r_1 - s_i^{-1}r_i)$. The bound given by the theorem 1 remains true with this method.

4.2.2 Remove the Second Column

There is an other way to make the previous attack independent of the size of the secret key, but this trick needs a slight modification of the LLL Reduction Algorithm ([Poh87]). Let the lattice L' spanned by the row vectors of the following matrix

$$M' = \begin{pmatrix} 1 & \alpha_2 & \dots & \alpha_n \\ 0 & \beta_2 & \dots & \beta_n \\ 0 & q & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & q \end{pmatrix}, \quad n > 2 \tag{8}$$

The matrix M' is the matrix M without the second column. As previously, we have that $v'_0 = (1, \kappa_2, \kappa_3, \dots, \kappa_n)$ is an element of the lattice L'. Indeed, there are $\lambda_2, \dots, \lambda_n \in \mathbb{Z}$ such that

$$(1, \mathbf{a}, \lambda_2, \dots, \lambda_n) \cdot M' = v'_0 \tag{9}$$

However, the row vectors of this matrix M' are not linearly independent. Thus, they do not form a basis of the lattice L' and the original LLL algorithm can not be applied directly. In this case, we use the MLLL algorithm ([Poh87]) which is a variant of LLL in which the input vectors can be linearly dependent, and has the same complexity as the LLL algorithm.

Remark 1. Note that the secret key must be read in the transformation vector $(1, \mathbf{a}, \lambda_2, \dots, \lambda_n)$ and not in the reduced basis. We could also have removed the first column, but experiments show that the required number of messages is greater in this case.

The lattice L' used in this case is different from the lattice L of theorem 1. The bound and the volume must be updated.

Lemma 1. *The volume of the lattice L' defined by the matrix M' given as (8) is equal to q^{n-2}.*

Proof. The sublattice S of L' spanned by the row vectors of the following matrix

$$\begin{pmatrix} 1 & \alpha_2 & \dots & \alpha_n \\ 0 & q & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & q \end{pmatrix}$$

is a n-dimensional lattice. The dimension of the lattice L' defined by the matrix M' given as (8) is equal to the dimension of its sublattice S, therefore S is a full-rank sublattice of L' (see [NV09]). We consider a lattice as a group and we have the classical relation between volume and index: $\mathrm{Vol}(S) = \mathrm{Vol}(L')[L' : S]$. Now, it is easy to see that $\mathrm{Vol}(S) = q^{n-1}$ and $[L' : S] = q$, from which we get $\mathrm{Vol}(L') = q^{n-2}$. $\square$

The following theorem is directly derived from this lemma.

Theorem 2. *Let n messages m_i ($i = 1,\dots,n$, $n > 2$) with associated signatures (r_i, s_i) such that the ephemeral keys $\mathbf{k}_i$ share a total of δ bits between the MSB and LSB (see Figure 1). Under the assumption 1, the secret key $\mathbf{a}$ can be computed in time $\mathscr{C}(n, \frac{1}{2}\log_2(n-1)+N)$ as soon as*

$$\delta \geq \frac{2N+(n-2)}{n} + \frac{1+\log_2(\pi e) - \log_2(\frac{n}{n-1})}{2}$$

The required number of shared bits δ is slightly larger with the lattice L' defined by the matrix M' given as (8) than the one given in the theorem 1. Experiments confirm this fact but the success rate is now independent of the secret key size (see section 5).

Remark 2. As a referee pointed out to us, this result may be reworded by CVP instead of SVP. This will be detailed in an extended version of this paper.

4.2.3 Weighted Euclidean Inner Product: In order to obtain the v_0 vector (7) (or similarly the v'_0 vector (9)) within a LLL-reduced basis, we can also use a weighted Euclidean inner product, to take advantage of the knowledge of the components size of the targeted vector. For example, we can take the following inner product of two vectors

$$\langle (x_0,\dots,x_n),(y_0,\dots,y_n)\rangle := \sum_{i=0}^{n} x_i y_i 2^{2(N-\lceil \log_2(v_{0,i})\rceil)}$$

during the LLL algorithm. In practice, this trick drastically reduces the required number of shared bits δ (see section 5).

Remark 3. Weights can be used without needing to change the norm. Indeed, it is equivalent to multiplying all columns of the lattice by the corresponding weight.

As previously, a bound can be obtained with Gaussian Heuristic.

Theorem 3. *Let n messages m_i ($i = 1,\dots,n$, $n > 2$) with associated signatures (r_i, s_i) such that the ephemeral keys $\mathbf{k}_i$ share a total of δ bits between the MSB and LSB. Under the assumption 1, the secret key $\mathbf{a}$ can be computed*

a. with the exhaustive search method in time $\delta\mathscr{C}(n+1, \frac{1}{2}\log_2(n)+\delta N)$ as soon as

$$\delta \geq \frac{N+(n-1)}{n} + \frac{(n+1)(1+\log_2(\pi e))}{2n} \tag{10}$$

b. with the lattice L' (8) in time $\mathscr{C}(n, \frac{1}{2}\log_2(n-1)+\delta N)$ as soon as

$$\delta \geq \frac{N+(n-2)}{n-1} + \frac{n(1+\log_2(\pi e))}{2(n-1)} \tag{11}$$

Proof. Let an integer $k \geq N-\delta$. We use a weighted Euclidean inner product such that each component of the targeted vector v_0 have the same size k (i.e. the i-th weight is equal to $k - \lceil \log_2(v_{0,i})\rceil$).

a. With the exhaustive search method, a close approximation of the vector v_0 (7) is computed. Its norm is given by $\|v_0\|^2 = \sum_{i=1}^{n+1} v_{0,i}^2 2^{2(k-\lceil \log_2(v_{0,i})\rceil)} \leq \sum_{i=1}^{n+1} 2^{2k} = 2^{2k+\log_2(n+1)}$ and the volume of the lattice (6) is

$$\mathrm{Vol}(L) = 2^k 2^{k-(N-\delta)}(q2^{k-(N-\delta)})^{(n-1)} \geq 2^{k(n+1)+n(\delta-1)-N+1}.$$

Using the Gaussian heuristic assumption, we have $2^{2k+\log_2(n+1)} \leq \frac{n+1}{2\pi e}(2^{k(n+1)+n(\delta-1)-N+1})^{\frac{2}{n+1}}$ which is equivalent to (10).

b. We apply the same method with the n-dimensional lattice L' (8) and the seek vector v'_0 (9). We obtain $\|v'_0\|^2 = \sum_{i=1}^{n} v'^2_{0,i} 2^{2(k-\lceil \log_2(v'_{0,i})\rceil)} \leq \sum_{i=1}^{n} 2^{2k} = 2^{2k+\log_2(n)}$ and $\mathrm{Vol}(L') = \frac{2^k(q2^{k-(N-\delta)})^{(n-1)}}{q} \geq 2^{n(k-1)+\delta(n-1)-N+2}$. The Gaussian heuristic assumption gives $2^{2k+\log_2(n)} \leq \frac{n}{2\pi e}(2^{n(k-1)+\delta(n-1)-N+2})^{\frac{2}{n}}$ which is equivalent to (11).

Note that both results are independent of the variable k. □

4.3 Blocks of Shared Bits

The previous attack can be generalized to the case of ephemeral keys sharing several blocks of bits. Thus, we now assume that we have n messages m_i $(i = 1,\ldots,n)$ with associated signatures (r_i,s_i) such that the ephemeral keys $\mathbf{k}_i$ share a total of δ bits dispatched between l blocks of bits. We denote by δ_i the number of bits of the i-th block $\mathbf{b}_i$ at position p_i (see Figure 2). For convenience we simplify the notation as follows: let $\underline{t} = (t_1,\ldots,t_l)$ be a l-tuple of integers then we set $2^{\underline{t}} = (2^{t_1},\ldots,2^{t_l})$ Then the ephemeral key $\mathbf{k}_i$ is of the form

$$\mathbf{k}_i = 2^{\underline{p}} \cdot \underline{\mathbf{b}} + 2^{\underline{t}} \cdot \underline{\mathbf{k}_i} \quad \text{for all } i = 1,\ldots,n \tag{12}$$

where $\underline{\mathbf{b}} = (\mathbf{b}_1,\ldots,\mathbf{b}_l)$ is the vector of shared bits blocks, with the position vector $\underline{p} = (p_1,\ldots,p_l)$ of the l blocks, and $\underline{\mathbf{k}_i} = (\mathbf{k}_{i,0},\ldots,\mathbf{k}_{i,l})$ the vector of no shared bits blocks at positions $\underline{t} = (t_0,\ldots,t_l)$. After stating that $t_0 := 0$ and $p_{l+1} := N$, it follows that for all $i = 1,\ldots,n$, and for all $j = 1,\ldots,l$, we must have

$$t_j = p_j + \delta_j, \quad \delta = \sum_j \delta_j, \quad 0 \leq \mathbf{b}_j < 2^{\delta_j} \quad \text{and} \quad 0 \leq \mathbf{k}_{i,j} < 2^{(p_{j+1}-t_j)}$$

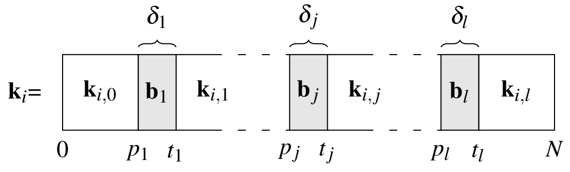


Fig. 2. Ephemeral keys

Note that the values of $\mathbf{k}_i$, $\mathbf{k}_{i,j}$ and $\mathbf{b}_j$ are all unknown. In the n signature equations (1), we substitute the $\mathbf{k}_i$ by (12) and we eliminate the common variable $\underline{\mathbf{b}}$, then we have

$$\begin{cases} (s_1^{-1}m_1 - s_2^{-1}m_2) + \mathbf{a}(s_1^{-1}r_1 - s_2^{-1}r_2) - \sum_{j=0}^{l} 2^{t_j}(\mathbf{k}_{1,j} - \mathbf{k}_{2,j}) \equiv 0 \pmod q \\ (s_1^{-1}m_1 - s_3^{-1}m_3) + \mathbf{a}(s_1^{-1}r_1 - s_3^{-1}r_3) - \sum_{j=0}^{l} 2^{t_j}(\mathbf{k}_{1,j} - \mathbf{k}_{3,j}) \equiv 0 \pmod q \\ \quad \vdots \qquad\qquad \vdots \qquad\qquad\qquad\qquad \vdots \qquad \vdots \\ (s_1^{-1}m_1 - s_n^{-1}m_n) + \mathbf{a}(s_1^{-1}r_1 - s_n^{-1}r_n) - \sum_{j=0}^{l} 2^{t_j}(\mathbf{k}_{1,j} - \mathbf{k}_{n,j}) \equiv 0 \pmod q \end{cases} \tag{13}$$

Let α_i, $\beta_i \in \mathbb{Z}$ and $\underline{\kappa_i}, \underline{t} \in \mathbb{Z}^l$ be such that

$$\begin{cases} \alpha_i := (s_1^{-1}m_1 - s_i^{-1}m_i) \bmod q \\ \beta_i := (s_1^{-1}r_1 - s_i^{-1}r_i) \bmod q \\ \underline{\kappa_i} := (\mathbf{k}_{1,1}, \ldots, \mathbf{k}_{1,l}) - (\mathbf{k}_{i,1}, \ldots, \mathbf{k}_{i,l}) \\ \underline{t} \;\; := (t_1, \ldots, t_l) \end{cases}$$

then (13) becomes

$$\begin{cases} \alpha_2 + \mathbf{a}\beta_2 - 2^{\underline{t}} \cdot \underline{\kappa_2} \equiv \mathbf{k}_{1,0} - \mathbf{k}_{2,0} \pmod q \\ \alpha_3 + \mathbf{a}\beta_3 - 2^{\underline{t}} \cdot \underline{\kappa_3} \equiv \mathbf{k}_{1,0} - \mathbf{k}_{3,0} \pmod q \\ \;\vdots \qquad \vdots \qquad\quad \vdots \qquad\qquad\quad \vdots \\ \alpha_n + \mathbf{a}\beta_n - 2^{\underline{t}} \cdot \underline{\kappa_n} \equiv \mathbf{k}_{1,0} - \mathbf{k}_{n,0} \pmod q \end{cases} \tag{14}$$

where $\mathbf{a}$ and $\underline{\kappa_i}$ are unknown, β_i and α_i are known. Embedding these equations into the lattice L spanned by the row vectors of the following basis matrix

$$M = \left(\begin{array}{c|c} & \alpha_2 \ldots \alpha_n \\ & \beta_2 \ldots \beta_n \\ \cline{2-2} I_{l(n-1)+2} & 2^{t_1} I_{(n-1)} \\ \cline{2-2} & \vdots \\ \cline{2-2} & 2^{t_l} I_{(n-1)} \\ \hline 0 & q I_{(n-1)} \end{array} \right) \tag{15}$$

we obtain that

$$v_0 = (1, \mathbf{a}, \underbrace{\mathbf{k}_{1,1} - \mathbf{k}_{2,1}, \ldots, \mathbf{k}_{1,1} - \mathbf{k}_{n,1}}_{\kappa_{i,1}}, \underbrace{\ldots}_{\kappa_{i,j}}, \underbrace{\mathbf{k}_{1,l} - \mathbf{k}_{2,l}, \ldots, \mathbf{k}_{1,l} - \mathbf{k}_{n,l}}_{\kappa_{i,l}}, \underbrace{\mathbf{k}_{1,0} - \mathbf{k}_{2,0}, \ldots, \mathbf{k}_{1,0} - \mathbf{k}_{n,0}}_{\alpha_i + \mathbf{a}\beta_i - 2^{\underline{t'}} \cdot \underline{\kappa_i} \bmod q}) \tag{16}$$

is an element of the lattice L. If we were able to find this vector v_0 in L, then we could recover the secret key $\mathbf{a}$.

Example 1. For instance, if we have only one block (*i.e.* $l = 1$) of δ shared bits in the middle, then the n ephemeral keys are of the form

$$\mathbf{k}_i = \mathbf{k}_{i,0} + 2^{p_1}\mathbf{b}_1 + 2^{t_1}\mathbf{k}_{i,1} \quad \text{for all } i = 1, \ldots, n$$

where

$$\delta = \delta_1, \quad t_1 = p_1 + \delta_1, \quad 0 \le \mathbf{k}_{i,0} < 2^{p_1}, \quad 0 \le \mathbf{k}_{i,1} < 2^{N-t_1} \quad \text{and} \quad 0 \le \mathbf{b}_1 < 2^{\delta_1}$$

and α_i, β_i, $\kappa_i \in \mathbb{Z}$ be such that

$$\begin{cases} \alpha_i := (s_1^{-1} m_1 - s_i^{-1} m_i) \bmod q \\ \beta_i := (s_1^{-1} r_1 - s_i^{-1} r_i) \bmod q \\ \kappa_i := \mathbf{k}_{1,1} - \mathbf{k}_{i,1} \end{cases}$$

In this case, (14) becomes

$$\begin{cases} \alpha_2 + \mathbf{a}\beta_2 + 2^{t_1}\kappa_2 \equiv \mathbf{k}_{1,0} - \mathbf{k}_{2,0} \pmod q \\ \alpha_3 + \mathbf{a}\beta_3 + 2^{t_1}\kappa_3 \equiv \mathbf{k}_{1,0} - \mathbf{k}_{3,0} \pmod q \\ \vdots \quad \vdots \quad \vdots \quad \vdots \\ \alpha_n + \mathbf{a}\beta_n + 2^{t_1}\kappa_n \equiv \mathbf{k}_{1,0} - \mathbf{k}_{n,0} \pmod q \end{cases}$$

and the lattice L is spanned by the row vectors of the following basis matrix

$$M = \left(\begin{array}{cc|ccc|ccc} 1 & 0 & 0 & \dots & 0 & \alpha_2 & \dots & \alpha_n \\ 0 & 1 & 0 & \dots & 0 & \beta_2 & \dots & \beta_n \\ \hline 0 & 0 & 1 & \dots & 0 & 2^{t_1} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 & 0 & \dots & 2^{t_1} \\ \hline 0 & 0 & 0 & \dots & 0 & q & \dots & 0 \\ \vdots & \vdots & \vdots & & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 0 & 0 & \dots & q \end{array}\right)$$

The targeted element of L is the vector

$$v_0 = (1, \mathbf{a}, \kappa_2, \dots, \kappa_n, \mathbf{k}_{1,0} - \mathbf{k}_{2,0}, \dots, \mathbf{k}_{1,0} - \mathbf{k}_{n,0}) = (1, \mathbf{a}, \kappa_2, \dots, \kappa_n, \lambda_2, \dots, \lambda_n) \cdot M$$

The proposed improvements of the section 4.2 can also be (directly) applied in the case of shared bits blocks. Especially, a weighted Euclidean inner product gives rise to the following estimations of the required number of shared bits in function of the number of blocks and the number of available messages.

Theorem 4. *Let n messages m_i ($i = 1, \dots, n$, $n > 2$) with associated signatures (r_i, s_i) such that the ephemeral keys $\mathbf{k}_i$ share a total of δ bits dispatched between l blocks of bits. Under the assumption 1, the secret key $\mathbf{a}$ can be computed*

a. with the exhaustive search method in time $\delta\mathscr{C}((l+1)(n-1)+2, \frac{1}{2}\log_2(n-1))$ as soon as

$$\delta \ge \frac{N + (n-1)}{n} + (1 + \log_2(\pi e))\frac{(l+1)(n-1)+2}{2n} \tag{17}$$

b. with the lattice L' obtained by removing the second column (8) in time $\mathscr{C}((l+1)(n-1)+1, \frac{1}{2}\log_2(n-1))$ as soon as

$$\delta \ge \frac{N + (n-2)}{n-1} + (1 + \log_2(\pi e))\frac{(l+1)(n-1)+1}{2(n-1)} \tag{18}$$

The proof of this theorem is almost similar as the one developed in Section 4.2 and is given in the Appendix B.

5 Experimental Results

In order to check the validity and the quality of the bounds on δ, we implemented the methods on the computational algebra system Magma V2.17-1 ([BCP97]). All the tests have essentially validated the Gaussian Heuristic assumption (assumption 1) and the fact that the first gap of the lattices (defined as λ_2/λ_1, see Annex A) is high enough so that the shortest vector can be computed with the LLL algorithm. Hence they show the effectiveness of our method. In the following, the length of q is fixed to $N = 160$.

5.1 Shared MSB and LSB

The figure 3 is the graph of the four bounds on δ given by the theorems 1, 2 and 3 in function of the number of messages. The table 2 gives more details by listing some theoretical minimal integer values of the necessary number of LSB/MSB shared bits δ for a given number of messages.

We conducted experiments of this attack when the ephemeral keys have their δ LSB in common. For the same reason as the one explained in [NS02], the results for the case of MSB or both MSB/LSB are not as good as in the LSB case, about one more bit being required. As the secret key is 160 bits long, we used the independent key size method by removing the corresponding column of the lattice (see section 4.2.2). Additionally, we use a weighted Euclidean inner product during the LLL algorithm phase which gives better results than the canonical inner product (more precisely, we use the inner product given as an example in section 4.2.3). The experiments are then conducted under the theorem 3b conditions. Note that we consider an attack to be successful when the secret

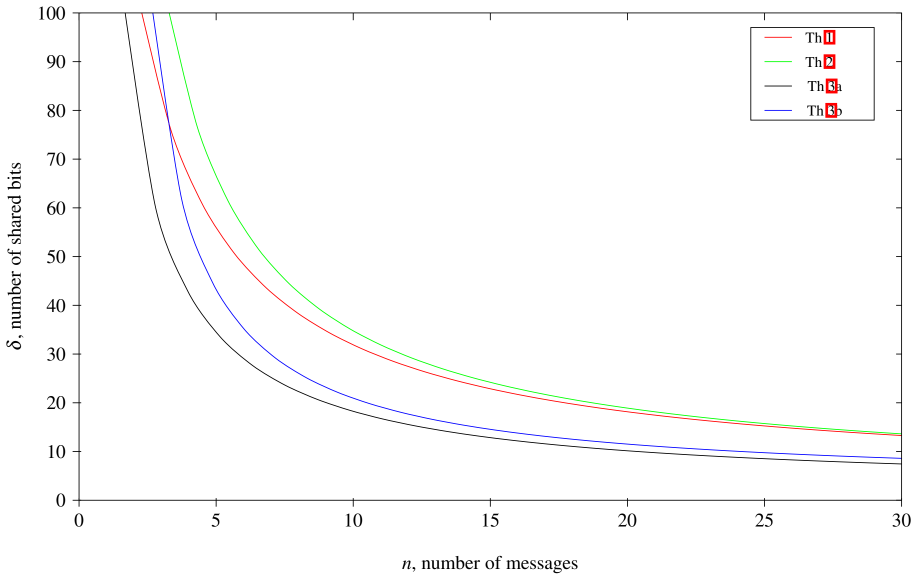


Fig. 3. Theoretical bounds of Theorems 1, 2 and 3 with $N = 160$

Table 2. Theoretical minimum for δ with $N = 160$

Bound on δ of theorem	n, number of messages																			
	3	4	5	6	7	8	9	10	15	20	30	40	50	60	70	80	90	100	200	∞
1	83	67	56	49	43	39	35	32	23	19	14	11	10	9	8	7	7	7	5	$\approx$3.05
2	109	83	67	56	49	43	39	35	25	19	14	11	10	9	8	8	7	7	5	$\approx$3.05
3a	57	44	36	30	27	24	21	20	14	12	9	8	7	6	6	6	5	5	4	$\approx$3.05
3b	84	57	44	36	30	27	24	21	15	12	9	8	7	6	6	6	5	5	4	$\approx$3.05

Table 3. Success rate of LSB attack of theorem 3b

δ	n, Number of messages																					
	40	50	60	70	80	90	100	110	120	130	140	150	160	170	180	190	200	250	300	400	500	600
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	8	10	35	56	91	99	99
2	0	0	0	0	0	2	6	12	24	33	42	58	63	73	80	85	100	100	100	100	100	100
3	0	2	19	34	60	74	82	94	96	97	99	99	99	99	99	100	100	100	100	100	100	100
4	34	76	90	99	99	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
5	96	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100	100
Time (s)	0.35	0.58	0.78	0.94	1.2	1.4	1.7	1.9	2.1	2.4	2.6	2.9	3.2	3.5	3.8	4.1	4.2	6.3	8.5	15	27	44

key is found, that is when the targeted vector $\pm v_0$ (9) is a vector of the reduced basis. For each δ and each n, we generated 100 tests and store the success rate in the table 3. To compare experimental values to the theoretical bound, the success rates corresponding to the theoretical minimal value of δ for a given number of messages are written in red.

First of all, the results show that we have a 100% success rate when δ is the bound (11) of theorem 3 and then we could say that they confirm that the assumptions we made are justified. Moreover, we observe that we can often go a few bits beyond the theoretical bound on δ. Then, the success rate gradually decreases to zero percent with δ.

Another interesting result is that the attack still works with only 1 or 2 shared bits ($\delta = 1, 2$) when there are enough messages. This result is particularly surprising considering that the theoretical limit is 3 and that the success rate can be higher than 90% (not to say 99% or 100% when $n > 500$).

Note also that the step of lattice reduction of this attack is very fast (always less than a minute, or even less than a second up to $n \approx 70$).

5.2 Blocks of Shared Bits

Following the bound (18) (better than the bound (17)) of theorem 4, the table 4 gives some theoretical minimal integer values of the necessary number of total shared bits δ in function of the number of blocks and the number of messages. Under theses conditions, we conducted a large number of experiments (100 tests for each δ and each n) whose results are summarized in table 5. Once more, we wrote in red the success rates corresponding to the theoretical minimal value of δ for a given number of messages.

Contrary to the case of shared LSB/MSB, we may have a success rate lower than 100% when δ was within the bound with the attack of blocks of shared bits (see Section 4.3). The reason is that the assumption of the theorem 4 may fail because of the occurrence of exceptionally short vectors. However, we observe that we can always go a few bits beyond the theoretical bound on δ keeping a good success rate.

We can also note that the computation time is longer than in the case LSB/MSB because of the larger dimension of lattices.

Table 4. Theoretical minimum for δ with theorem 4b

Number of blocks l	n, number of messages													
	3	4	5	6	7	8	9	10	15	20	30	40	50	∞
1	86	59	46	38	32	29	26	23	17	14	11	10	9	6
2	88	61	48	40	34	31	28	26	19	16	13	12	11	8
3	90	63	50	42	37	33	30	28	21	18	15	14	13	10
10	105	78	64	56	51	47	44	42	36	32	30	28	27	24
20	125	98	85	77	71	67	65	62	56	53	50	49	48	44
30	145	119	105	97	92	88	85	83	76	73	71	69	68	65

Table 5. Success rate of theorem 4b with one block

δ	n, number of messages															
	13	14	15	16	17	18	19	20	30	40	50	60	70	80	90	100
4	0	0	0	0	0	0	0	0	0	0	6	13	19	28	26	49
5	0	0	0	0	0	0	0	0	0	19	32	49	67	78	82	77
6	0	0	0	0	0	0	0	0	13	49	78	89	90	94	100	99
7	0	0	0	0	0	0	0	1	63	89	96	99	97	99	97	96
8	0	0	0	0	2	1	5	20	93	99	99	100	99	98	98	98
9	0	0	0	1	12	21	49	59	99	100	99	99	100	99	100	99
10	0	1	1	21	60	82	95	94	100	100	100	100	99	99	100	100
Time (s)	0.08	0.10	0.13	0.16	0.20	0.23	0.28	0.3	0.8	1.4	2.1	3.0	4.1	5	6	6.8

5.3 Random Number Generator Tests

In the scenario with malicious PRNG, we verified that a defect is experimentally undetectable by conventional tests. Indeed, a 8 GByte bit sequence from the AES_OFB random number generator in Dieharder ([Bro11]), manipulated to contain enough implicit information, was used as input for the Dieharder test suite. More precisely, sequences of some randomly selected bits are repeated in a predictable way. For instance, a sequence of 4 bits can be repeated 100 times following a predictable pattern in a random sequence corresponding to 2^{10} ephemeral keys. The pattern that describes the positions of corrupted nonces (i.e. ephemeral keys sharing some bits) can be, for example, a function of the position of the first corrupted nonce. Therefore, in this case we need an additional step containing an exhaustive search to find the first corrupted nonce (of complexity of about 2^{10} with this example). All tests of the two referenced statistical test suites: Dieharder statistical test suite ([Bro11]) and the NIST statistical test suite (STS) ([RSN+10]) have shown a random behavior at a high confidence level, our manipulations being then unnoticed when the number of shared bits matches the number of corrupted nonces to have a 100% success rate (see Table 3). These experiments remind that these tests are not a proof of randomness even though they are common tools for initial validation. Finally, it has been shown that an exploitable bias is currently undetectable by conventional statistical tests.

6 Further Developments

Throughout this work, we assumed that all ephemeral keys used in the attack shared a same block of bits. In this scenario, by the pigeonhole principle, our attack needs, in the worst case, $2^{\delta} + 1$ samples before obtaining only two signatures that have ephemeral keys with δ bits in common. However, from a practical perspective, we would like to use all the signatures generated by a signer, which is not possible for the moment. Assuming, as in [LPS04], that an attacker can determine, in practice, some relation amongst the bits of the secret ephemeral keys rather than their specific values, we present below how to solve this problem by slightly adapting the lattices of our method.

In this context, our attack can be naturally extended to the more general case where each ephemeral key $\mathbf{k}_i$ $(i = 1 \dots n)$ shares δ bits with at least one other key $\mathbf{k}_j$ $(i \neq j)$ and not necessarily with all of them. For instance, we look at shared MSB/LSB, with the same notation as in section 4.1. For two fixed positions t and t', we can take the partition P of the set of all ephemeral keys corresponding to the equivalence relation $\mathscr{R}_{t,t'}$, defined such that related ephemeral keys share the first t MSB and the last t' LSB (which represent a total of δ bits). In a given equivalence class $[\mathbf{k}_j]$, if we have $\#[\mathbf{k}_j] \geq 2$ then we can apply the method of Section 4.1. For each class $[\mathbf{k}_j]$, we obtain a system of $\#[\mathbf{k}_j] - 1$ modular equations as (5) with

$$\forall \mathbf{k}_i \in [\mathbf{k}_j] \text{ s.t. } i \neq j, \begin{cases} \alpha_i := 2^{-t}(s_j^{-1} m_j - s_i^{-1} m_i) \mod q \\ \beta_i := 2^{-t}(s_j^{-1} r_j - s_i^{-1} r_i) \quad \mod q \\ \kappa_i := \tilde{\mathbf{k}_j} - \tilde{\mathbf{k}_i} \end{cases}$$

The set of all common solutions of the $\#P = \#\left(\{\mathbf{k}_i, i = 1..n\}/R_{t,t'}\right)$ systems described as above forms a lattice similar to (6) of dimension equal to $n - \#P$.

In the same way, other shared bits in other positions (i.e. when t and t' are not fixed) can be exploited by expanding the lattice with the corresponding columns. Also the same improvement can be developped in the case of shared bits blocks in the middle. All these further developments will be detailed in an extended version of this paper. More generally, we could also imagine other forms of implicit information, i.e. an other relationship than just the equality between bits. For instance, an interesting open question is whether we can exploit inequalities or simple relationships between unknown bits.

Acknowledgments. This work has greatly benefited from highly relevant comments of anonymous referees. The authors would like to thank them sincerely for their suggestions that improved this paper. They also thank Olivier Orcière for all his constructive comments on a preliminary version of this paper. Finally, they thank Martin Albrecht for presenting this paper at the SAC2012 conference.

References

[AD93] Adleman, L.M., DeMarrais, J.: A Subexponential Algorithm for Discrete Logarithms over All Finite Fields. In: Stinson, D.R. (ed.) CRYPTO 1993. LNCS, vol. 773, pp. 147–158. Springer, Heidelberg (1994)

[ADH94] Adleman, L.M., DeMarrais, J., Huang, M.-D.A.: A Subexponential Algorithm for Discrete Logarithms over the Rational Subgroup of the Jacobians of Large Genus Hyperelliptic Curves over Finite Fields. In: Huang, M.-D.A., Adleman, L.M. (eds.) ANTS 1994. LNCS, vol. 877, pp. 28–40. Springer, Heidelberg (1994)

[ADH99] Adleman, L.M., DeMarrais, J., Huang, M.-D.A.: A Subexponential Algorithm for Discrete Logarithms over Hyperelliptic Curves of Large Genus over GF(q). Theoretical Computer Science 226(1-2), 7–18 (1999)

[Ajt98] Ajtai, M.: The shortest vector problem in l_2 is *np*-hard for randomized reductions (extended abstract). In: Proceedings of the 30th Symposium on the Theory of computing (STOC 1998), pp. 10–19. ACM Press (1998)

[Ajt06] Ajtai, M.: Generating random lattices according to the invariant distribution. draft (March 2006)

[BCP97] Bosma, W., Cannon, J., Playoust, C.: The MAGMA algebra system: the user language. J. Symb. Comput. 24, 235–265 (1997)

[BGM97] Bellare, M., Goldwasser, S., Micciancio, D.: "Pseudo-random" Number Generation within Cryptographic Algorithms: The DSS Case. In: Kaliski Jr., B.S. (ed.) CRYPTO 1997. LNCS, vol. 1294, pp. 277–291. Springer, Heidelberg (1997)

[Bro11] Brown, R.G.: DieHarder: A Random Number Test Suite. C program archive dieharder, version 3.29.4b (2011), `http://www.phy.duke.edu/~rgb/General/dieharder.php`

[Cop96] Coppersmith, D.: Finding a Small Root of a Bivariate Integer Equation; Factoring with High Bits Known. In: Maurer, U.M. (ed.) EUROCRYPT 1996. LNCS, vol. 1070, pp. 178–189. Springer, Heidelberg (1996)

[ElG85] El Gamal, T.: A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. In: Blakely, G.R., Chaum, D. (eds.) CRYPTO 1984. LNCS, vol. 196, pp. 10–18. Springer, Heidelberg (1985)

[FIP94] FIPS. Digital Signature Standard (DSS). National Institute of Standards and Technology, NIST (1994)

[FIP09] FIPS. Digital Signature Standard (DSS). pub-NIST, pub-NIST:adr (2009)

[FMR10] Faugère, J.-C., Marinier, R., Renault, G.: Implicit Factoring with Shared Most Significant and Middle Bits. In: Nguyen, P.Q., Pointcheval, D. (eds.) PKC 2010. LNCS, vol. 6056, pp. 70–87. Springer, Heidelberg (2010)

[GJQ97] Gillet, A., Joye, M., Quisquater, J.-J.: Cautionary note for protocols designers: Security proof is not enough. In: Orman, H., Meadows, C. (eds.) DIMACS Workshop on Design and Formal Verification of Security Protocols (January 1997)

[HGS01] Howgrave-Graham, N., Smart, N.P.: Lattice Attacks on Digital Signature Schemes. Des. Codes Cryptography 23(3), 283–290 (2001)

[JMV01] Johnson, D., Menezes, A., Vanstone, S.A.: The Elliptic Curve Digital Signature Algorithm (ECDSA). Intern. J. of Information Security 1(1), 36–63 (2001)

[Len87] Lenstra, H.W.: Factoring Integers with Elliptic Curves. The Annals of Mathematics 126(3), 649–673 (1987)

[LL93] Lenstra, A.K., Lenstra, H.W.: The Development of the Number Field Sieve. Lecture Notes in Mathematics, vol. 1554. Springer, Berlin (1993)

[LLL82] Lenstra, A., Lenstra, H., Lovász, L.: Factoring polynomials with rational coefficients. Mathematische Annalen 261(4), 515–534 (1982)

[LPS04] Leadbitter, P.J., Page, D.L., Smart, N.P.: Attacking DSA Under a Repeated Bits Assumption. In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 428–440. Springer, Heidelberg (2004)

[MNS01] El Mahassni, E., Nguyên, P.Q., Shparlinski, I.E.: The Insecurity of Nyberg-Rueppel and Other DSA-Like Signature Schemes with Partially Known Nonces. In: Silverman, J.H. (ed.) CaLC 2001. LNCS, vol. 2146, pp. 97–109. Springer, Heidelberg (2001)

[MR09] May, A., Ritzenhofen, M.: Implicit Factoring: On Polynomial Time Factoring Given Only an Implicit Hint. In: Jarecki, S., Tsudik, G. (eds.) PKC 2009. LNCS, vol. 5443, pp. 1–14. Springer, Heidelberg (2009)

[NNTW05] Naccache, D., Nguyên, P.Q., Tunstall, M., Whelan, C.: Experimenting with Faults, Lattices and the DSA. In: Vaudenay, S. (ed.) PKC 2005. LNCS, vol. 3386, pp. 16–28. Springer, Heidelberg (2005)

[NS02] Nguyên, P.Q., Shparlinski, I.: The Insecurity of the Digital Signature Algorithm with Partially Known Nonces. Journal of Cryptology 15, 151–176 (2002)

[NS03] Nguyên, P.Q., Shparlinski, I.: The Insecurity of the Elliptic Curve Digital Signature Algorithm with Partially Known Nonces. Designs, Codes and Cryptography 30(2), 201–217 (2003)

[NS05] Nguyên, P.Q., Stehlé, D.: Floating-Point LLL Revisited. In: Cramer, R. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 215–233. Springer, Heidelberg (2005)

[NV09] Nguyên, P.Q., Vallée, B.: Hermite's Constant and Lattice Algorithms. Information Security and Cryptography. Springer (2009)

[Poh87] Pohst, M.: A Modification of the LLL Reduction Algorithm. Journal of Symbolic Computation 4, 123–127 (1987)

[Pol78] Pollard, J.M.: Monte Carlo methods for index computation (mod *p*) 32, 918–924 (1978)

[Pol00] Pollard, J.M.: Kangaroos, Monopoly and discrete logarithms 13, 437–447 (2000)

[Pom84] Pomerance, C.: The Quadratic Sieve Factoring Algorithm. In: Beth, T., Cot, N., Ingemarsson, I. (eds.) EUROCRYPT 1984. LNCS, vol. 209, pp. 169–182. Springer, Heidelberg (1985)

[Pou09] Poulakis, D.: Some lattices attacks on dsa and ecdsa. Cryptology ePrint Archive, Report 2009/363 (2009), http://eprint.iacr.org/

[RS86] Rivest, R.L., Shamir, A.: Efficient Factoring Based on Partial Information. In: Pichler, F. (ed.) EUROCRYPT 1985. LNCS, vol. 219, pp. 31–34. Springer, Heidelberg (1986)

[RSN+10] Rukhin, A., Soto, J., Nechvatal, J., Smid, M., Barker, E., Leigh, S., Levenson, M., Vangel, M., Banks, D., Heckert, A., Dray, J., Vo, S.: A Statistical Test Suite of Random and Pseudorandom Number Generators for Cryptographic Applications. Tech. rep., National Institute of Standards and Technology (NIST), Special Publication 800-22 Revision 1a (2010), http://csrc.nist.gov/groups/ST/toolkit/rng/documentation_software.html

[Sch90] Schnorr, C.-P.: Efficient Identification and Signatures for Smart Cards. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 239–252. Springer, Heidelberg (1990)

[Sha71] Shanks, D.: Class number, a theory of factorization, and genera. In: Proceedings of Symposia in Pure Mathematics, vol. 20, pp. 415–440. American Mathematical Society, Providence (1971)

[Sho97] Shoup, V.: Lower Bounds for Discrete Logarithms and Related Problems. In: Fumy, W. (ed.) EUROCRYPT 1997. LNCS, vol. 1233, pp. 256–266. Springer, Heidelberg (1997)

[Sko05] Skorobogatov, S.P.: Semi-invasive attacks - A new approach to hardware security analysis. PhD thesis, University of Cambridge (2005)

[SM09] Sarkar, S., Maitra, S.: Further Results on Implicit Factoring in Polynomial Time. Advances in Mathematics of Communications 3(2), 205–217 (2009)

[Tak06a] Takashima, K.: Practical Application of Lattice Basis Reduction Algorithm to Side-Channel Analysis on (EC)DSA. IEICE Transactions 89-A(5), 1255–1262 (2006)

[Tak06b] Takashima, K.: Practical Modifications of Leadbitter et al.'s Repeated-Bits Side-Channel Analysis on (EC)DSA. In: Song, J.-S., Kwon, T., Yung, M. (eds.) WISA 2005. LNCS, vol. 3786, pp. 259–270. Springer, Heidelberg (2006)

[Tes01] Teske, E.: Square-Root Algorithms For The Discrete Logarithm Problem (a Survey). In: Public Key Cryptography and Computational Number Theory, pp. 283–301. Walter de Gruyter (2001)

A Common Results on Lattice

In this section, we state common results on lattices that are used throughout this paper. Readers interested in getting more details and proofs can refer to [NV09].

An integer lattice L is a discrete additive subgroup of $\mathbb{Z}^n$. It can be generated from a basis of d independent vectors $(b_1,\dots,b_d)$ of $\mathbb{Z}^n$ by linear combinations with integer coefficients. A lattice may be described by many different bases. All the bases are then related by an unimodular transformation. The integer d is called the dimension of L. If $d = n$ then L is said to be full-rank.

Definition 1. *The Gram determinant of $b_1,\dots,b_d \in \mathbb{R}^n$ denoted by $\Delta(b_1,\dots,b_d)$, is the determinant of the $d \times d$ Gram matrix $(\langle b_i, b_j\rangle)_{1\le i,j\le d}$.*

Definition 2. *The volume of L is defined by $\mathrm{Vol}(L) = \Delta(b_1,\dots,b_d)^{1/2}$. In other words, it is the d-dimensional volume of the parallelepiped spanned by the vectors of a basis.*

The dimension and the volume are independent of the choice of this basis.

The volume of a lattice is easy to compute with Definition 2 if at least one explicit basis is known. But, in this article, we also need a way to compute the volume of lattices spanned by a set of linearly dependent vectors. In this case, the following results on sublattices is very useful.

Definition 3. *A sublattice of L is a lattice M included in L. Clearly, the sublattices of L are the subgroups of L.*

Lemma 2 ([NV09]). *A sublattice M of L is full-rank if and only if the group index $[L:M]$ is finite, in which case we have*

$$\mathrm{Vol}(M) = \mathrm{Vol}(L) \times [L:M].$$

We also need the following results about lattice basis reduction. More particularly, we require a way to provide a precise estimation of the expected length of the shortest vector, which is called the Gaussian heuristic, and a way to get an approximation of this small vector in polynomial time, which is done by using the LLL algorithm ([LLL82]).

Definition 4. *For $1 \le r \le d$, let $\lambda_r(L)$ be the least real number such that there exists at least r linearly independent vectors of L of euclidean norm smaller or equal to $\lambda_r(L)$. We call $\lambda_1(L),\dots,\lambda_d(L)$ the d minima of L and we call $g(L) = \lambda_2(L)/\lambda_1(L)$ the gap of L.*

Theorem 5 (LLL [LLL82]). *Let L be a d-dimensional lattice of $\mathbb{Z}^n$ given by a basis $(\mathbf{b_1},\dots,\mathbf{b_d})$. Then LLL algorithm computes a reduced basis $(\mathbf{v_1},\dots,\mathbf{v_d})$ that approximates the shortest vector of L within an exponential factor:*

$$\|\mathbf{v_1}\| \le 2^{\frac{d-1}{4}}\,\mathrm{Vol}(L)^{\frac{1}{d}}$$

The running time of Nguyen and Stehlé's version is $\mathcal{O}(d^5(d+\log B)\log B)$ where $B = \max_i(\|\mathbf{b_i}\|)$, see [NS05].

The time complexity of computing a shortest vector of L (which is a NP-Hard problem [Ajt98]) is denoted here by $\mathcal{C}(d,B)$.

Theorem 6 (Gaussian heuristic [Ajt06]). *Let L be a random d-dimensional lattice of $\mathbb{Z}^n$. Then, with overwhelming probability, all the minima of L are asymptotically close to:*

$$\sqrt{\frac{d}{2\pi e}}\mathrm{Vol}(L)^{\frac{1}{d}}$$

Thus, it is common practice to assume that if a vector $v \in L$ is shorter than the Gaussian heuristic $\lambda_1(L) \approx \sqrt{\frac{d}{2\pi e}}\mathrm{Vol}(L)^{\frac{1}{d}}$ applied to the d-dimensional lattice L then it is the shortest vector of L. Moreover, when the gap of L is high enough, this vector can be found in an LLL-reduced basis of L. For lattices proposed in this article, this essential and common assumption is confirmed by experimental results of section 5 and seems to be true in practice.

B Proof of Theorem 4

Proof. Let an integer $k \geq N - \delta$. We use a weighted Euclidean inner product such that each component of the seek vector v_0 have the same size k (i.e. the i-th weight is equal to $k - \lceil \log_2(v_{0,i}) \rceil$).

a. With the exhaustive search method: first note that the dimension of the lattice L defined by (15) is

$$\dim(L) = (l+1)(n-1)+2$$

A close approximation of the norm of vector v_0 (16) is then computed:

$$\|v_0\|^2 = \sum_{i=1}^{\dim(L)} v_{0,i}^2 2^{2(k-\lceil \log_2(v_{0,i})\rceil)} \leq \sum_{i=1}^{\dim(L)} 2^{2k} = 2^{2k}((l+1)(n-1)+2)$$

Next, the volume of the lattice (15) is

$$\mathrm{Vol}(L) = 2^k 2^{k-(N-\delta)}(q2^{k-p_1})^{(n-1)} \prod_{j=1}^{l} 2^{(k-(p_{j+1}-t_j))(n-1)}$$

but, we have

$$\begin{aligned}
&(k - p_1)(n-1) + \textstyle\sum_{j=1}^{l}(k-(p_{j+1}-t_j))(n-1)\\
&= (n-1)(k-p_1+\textstyle\sum_{j=1}^{l}(k-p_{j+1}+p_j+\delta_j))\\
&= (n-1)(k-p_1+lk-\textstyle\sum_{j=1}^{l}p_{j+1}+\sum_{j=1}^{l}p_j+\sum_{j=1}^{l}\delta_j)\\
&= (n-1)(k(l+1)-p_{l+1}+\delta)\\
&= (n-1)(k(l+1)-N+\delta)
\end{aligned}$$

then

$$\mathrm{Vol}(L) = q^{n-1}2^k 2^{k-(N-\delta)}2^{(n-1)(k(l+1)-N+\delta)} \geq 2^{k((l+1)(n-1)+2)+n(\delta-1)-N+1}$$

Using the Gaussian heuristic assumption, we have

$$2^{2k+\log_2(\dim(L))} \leq \frac{\dim(L)}{2\pi e}(2^{k\dim(L)+n(\delta-1)-N+1})^{\frac{2}{\dim(L)}}$$

which is equivalent to

$$\delta \geq \frac{N+(n-1)}{n} + (1+\log_2(\pi e))\frac{\dim(L)}{2n}$$

b. Same computation with the lattice L' obtained by removing the second column of (15). The dimension of L' is equal to $(l+1)(n-1)+1$. Then a close approximation of the norm of vector v_0' is

$$\|v_0'\|^2 \leq 2^{2k}((l+1)(n-1)+1)$$

Next, the volume of L' is

$$\mathrm{Vol}(L') = 2^k q^{n-2}(2^{k-p_1})^{(n-1)} \prod_{j=1}^{l} 2^{(k-(p_{j+1}-t_j))(n-1)} \geq 2^{k((l+1)(n-1)+1)+(n-1)\delta-N-n+2}$$

Using the Gaussian heuristic assumption, we have

$$2^{2k}\dim(L') \leq \frac{\dim(L')}{2\pi e}(2^{k\dim(L')+(n-1)\delta-N-n+2})^{\frac{2}{\dim(L')}}$$

which is equivalent to

$$\delta \geq \frac{N+(n-2)}{n-1} + (1+\log_2(\pi e))\frac{\dim(L')}{2(n-1)} \qquad \square$$

Lattice Reduction for Modular Knapsack[*]

Thomas Plantard, Willy Susilo, and Zhenfei Zhang

Centre for Computer and Information Security Research
School of Computer Science & Software Engineering (SCSSE)
University of Wollongong, Australia
{thomaspl,wsusilo,zz920}@uow.edu.au

Abstract. In this paper, we present a new methodology to adapt any kind of lattice reduction algorithms to deal with the modular knapsack problem. In general, the modular knapsack problem can be solved using a lattice reduction algorithm, when its density is low. The complexity of lattice reduction algorithms to solve those problems is upper-bounded in the function of the lattice dimension and the maximum norm of the input basis. In the case of a low density modular knapsack-type basis, the weight of maximum norm is mainly from its first column. Therefore, by distributing the weight into multiple columns, we are able to reduce the maximum norm of the input basis. Consequently, the upper bound of the time complexity is reduced.

To show the advantage of our methodology, we apply our idea over the floating-point LLL (L^2) algorithm. We bring the complexity from $O(d^{3+\varepsilon}\beta^2 + d^{4+\varepsilon}\beta)$ to $O(d^{2+\varepsilon}\beta^2 + d^{4+\varepsilon}\beta)$ for $\varepsilon < 1$ for the low density knapsack problem, assuming a uniform distribution, where d is the dimension of the lattice, β is the bit length of the maximum norm of knapsack-type basis.

We also provide some techniques when dealing with a principal ideal lattice basis, which can be seen as a special case of a low density modular knapsack-type basis.

Keywords: Lattice Theory, Lattice Reduction, Knapsack Problem, LLL, Recursive Reduction, Ideal Lattice.

1 Introduction

To find the shortest non-zero vector within an arbitrary lattice is an NP-hard problem [1]. Moreover, till now there is no polynomial algorithm that finds a vector in the lattice that is polynomially close to the shortest non-zero vector. However, there exist several algorithms, for example, LLL [14] and L^2 [19], running in polynomial time in d and β, where d is the dimension of the lattice, and β is the bit length of the maximum norm of input basis, that find vectors with exponential approximation (in d) to the shortest non-zero vector. Indeed, in some lattice based cryptography/cryptanalysis, it may not be necessary to recover the exact shortest non-zero vector, nor a polynomially close one. Finding one with

[*] This work is supported by ARC Future Fellowship FT0991397.

exponential distance to the shortest one is already useful, for instance, to solve a low density knapsack problem or a low density modular knapsack problem.

Definition 1 (Knapsack Problem). *Let $\{X_1, X_2, \dots, X_d\}$ be a set of positive integers. Let $c = \sum_1^d s_i X_i$, where $s_i \in \{0,1\}$. A knapsack problem is given $\{X_i\}$ and c, find each s_i.*

The density of a knapsack, denoted by ρ, is d/β, where β is the maximum bit length of X_i-s.

Definition 2 (Modular Knapsack Problem). *Let $\{X_0, X_1, \dots, X_d\}$ be a set of positive integers. Let $c = \sum_1^d s_i X_i \bmod X_0$, where $s_i \in \{0,1\}$. A modular knapsack problem is given $\{X_i\}$ and c, find each s_i.*

The knapsack problem is also known as the subset sum problem [12]. When $\sum s_i \ll d$, it becomes a sparse subset sum problem (SSSP). The decisional version of the knapsack problem is NP-complete [9]. However, if its density is too low, there is an efficient reduction to the problem of finding the shortest vector from a lattice (refer to [13,21,3]).

In this paper, we deal with a (modular) knapsack problem assuming a uniform distribution, i.e., X_i-s are uniformly randomly distributed.

We refer to B_K as the *knapsack-type basis*, and B_M as the *modular knapsack-type basis*. In the rest of the paper, for simplicity, we focus on knapsack-type basis, although the adoption over a modular knapsack-type basis is straightforward.

$$B_K = \begin{pmatrix} X_1 & 1 & 0 & \dots & 0 \\ X_2 & 0 & 1 & \dots & 0 \\ X_3 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ X_d & 0 & 0 & \dots & 1 \end{pmatrix}, \qquad B_M = \begin{pmatrix} X_0 & 0 & 0 & \dots & 0 \\ X_1 & 1 & 0 & \dots & 0 \\ X_2 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ X_d & 0 & 0 & \dots & 1 \end{pmatrix}.$$

We also consider a *principal ideal lattice basis*. A principal ideal lattice is an ideal lattice that can be represented by two integers. This type of lattice enables important applications, for instance, constructing fully homomorphic encryption schemes with smaller key size (see [5,26] for an example of this optimization).

A basis of a principal ideal lattice (see Section 5) maintains a similar form of modular knapsack basis, with $X_i = -\alpha^i \bmod X_0$ for $i \geq 1$, where α is the root for the principal ideal. The security of the corresponding cryptosystem is based on the complexity of reducing this basis.

In general, to solve any of the above problems using lattice, one always start by performing an LLL reduction on a lattice $\mathcal{L}(B_K)$ ($\mathcal{L}(B_M)$, respectively). Then depending on the type of problem and the result the LLL algorithm produces, one may perform stronger lattice reduction (BKZ [24,7,2] for example) and/or use enumeration techniques such as Kannan SVP solver [8].

To date, the complexity of best LLL reduction algorithms for the above three type of basis is upper bounded by $O(d^{3+\varepsilon}\beta^2 + d^{4+\varepsilon}\beta)$ [19], although heuristically, one is able to obtain $O(d^{2+\varepsilon}\beta^2)$ in practice when ρ is significantly smaller than 1 [20].

Our Contribution: We propose a new methodology to reduce low density modular knapsack-type basis using LLL-type algorithms. Instead of reducing the whole knapsack-type basis directly, we pre-process its sub-lattices, and therefore, the weight of X_i-s is equally distributed into several columns and the reduction complexity is thereafter reduced. Although the idea is somewhat straightforward, the improvement is very significant.

Table 1. Comparison of time complexity

Algorithms	Time Complexity
LLL[14]	$O(d^{5+\varepsilon}\beta^{2+\varepsilon})$
LLL for knapsack	$O(d^{4+\varepsilon}\beta^{2+\varepsilon})$
L^2 [19]	$O(d^{4+\varepsilon}\beta^2 + d^{5+\varepsilon}\beta)$
L^2 for knapsack[19]	$O(d^{3+\varepsilon}\beta^2 + d^{4+\varepsilon}\beta)$
$\tilde{\text{L}}^1$ [22]	$O(d^{\omega+1+\varepsilon}\beta^{1+\varepsilon} + d^{5+\varepsilon}\beta)$
Our rec-L^2	$O(d^{2+\varepsilon}\beta^2 + d^{4+\varepsilon}\beta)$
Our rec-$\tilde{\text{L}}^1$	$O(d^{\omega}\beta^{1+\varepsilon} + d^{4+\varepsilon}\beta)$

Table 1 shows a time complexity comparison between our algorithms and the existing algorithms. However, we note that the complexities of all the existing algorithms are in worst-case, or in another words, for any type of basis, the algorithms will terminate in the corresponding time. In contrast, in our algorithm, we assume a uniform distribution, and therefore, the complexity given for our recursive reduction algorithms is the upper bound following this assumption. Nevertheless, we note that such an assumption is quite natural in practice.

Our result is also applicable to a principal ideal lattice basis. In addition, we provide a technique that further reduces the time complexity. Note that our technique does not affect the asymptotic complexity as displayed in Table 1.

Paper Organization: In the next section, we review some related area to this research. In Section 3, we propose our methodology to deal with low density knapsack-type basis, introduce our recursive reduction algorithm, and analyze its complexity. Then, we apply our method to L^2 and compare its complexity with non-modified L^2 in Section 4. In Section 5, we analyze the special case of the principal ideal lattice basis. Finally, the last section concludes this paper.

2 Background

2.1 Lattice Basics

In this subsection, we review some concepts of the lattice theory that will be used throughout this paper. The lattice theory, also known as the geometry of numbers, was introduced by Minkowski in 1896 [17]. We refer readers to [15,16] for a more complex account.

Definition 3 (Lattice). *A lattice $\mathcal{L}$ is a discrete sub-group of $\mathbb{R}^n$, or equivalently the set of all the integral combinations of $d \leq n$ linearly independent vectors over $\mathbb{R}$.*

$$\mathcal{L} = \mathbb{Z}b_1 + \mathbb{Z}b_2 + \cdots + \mathbb{Z}b_d, b_i \in \mathbb{R}^n$$

$B = (b_1, \ldots, b_d)$ is called a basis of $\mathcal{L}$ and d is the dimension of $\mathcal{L}$, denoted as $\dim(\mathcal{L})$. *$\mathcal{L}$ is a full rank lattice if d equals to n.*

For a given lattice $\mathcal{L}$, there exists an infinite number of basis. However, its determinant (see Definition 4) is unique.

Definition 4 (Determinant). *Let $\mathcal{L}$ be a lattice. Its determinant, denoted as* $\det(\mathcal{L})$*, is a real value, such that for any basis B of $\mathcal{L}$,* $\det(\mathcal{L}) = \sqrt{\det(B \cdot B^T)}$, *where B^T is the transpose of B.*

Definition 5 (Successive Minima). *Let $\mathcal{L}$ be an integer lattice of dimension d. Let i be a positive integer. The i-th successive minima with respect to $\mathcal{L}$, denoted by λ_i, is the smallest real number, such that there exist i non-zero linear independent vectors $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_i \in \mathcal{L}$ with*

$$\|v_1\|, \|v_2\|, \ldots, \|\boldsymbol{v}_i\| \leq \lambda_i.$$

The i-th minima of a random lattice (as defined in Theorem 1) is estimated by:

$$\lambda_i(\mathcal{L}) \sim \sqrt{\frac{d}{2\pi e}} \det(\mathcal{L})^{\frac{1}{d}}. \tag{1}$$

Definition 6 (Hermite Factor). *Let $B = (b_1, \ldots, b_d)$ a basis of $\mathcal{L}$. The Hermite factor with respect to B, denoted by $\gamma(B)$, is defined as* $\frac{\|b_1\|}{\det(\mathcal{L})^{\frac{1}{d}}}$.

Note that Hermite factor indicates the quality of a reduced basis.
Additionally, following the result of [6]:

Theorem 1 (Random Lattice). *Let B be a modular knapsack-type basis constructing from a modular knapsack problem given by $\{X_i\}$. $\mathcal{L}(B)$ is a random lattice, if $\{X_i\}$ are uniformly distributed.*

2.2 Lattice Reduction Algorithms

In 1982, Lenstra, Lenstra and Lovasz [14] proposed an algorithm, known as LLL, that produces an LLL-reduced basis for a given basis. For a lattice $\mathcal{L}$ with dimension d, and a basis B, where the norm of all spanning vectors in B is $\leq 2^\beta$, the worst-case time complexity is polynomial $O(d^{5+\varepsilon}\beta^{2+\varepsilon})$. Moreover, it is observed in [20] that in practice, LLL seems to be much more efficient in terms of average time complexity.

In 2005, Nguyen and Stehlé [19] proposed an improvement of LLL, which is the first variant whose worst-case time complexity is quadratic with respect to β.

This algorithm is therefore named L^2. This algorithm makes use of floating-point arithmetics, hence, the library that implements L^2 is sometimes referred to as *fplll* [23]. It terminates with a worst-case time complexity of $O(d^{4+\varepsilon}\beta^2 + d^{5+\varepsilon}\beta)$ for any basis. For a knapsack-type basis, it is proved that L^2 terminates in $O(d^{3+\varepsilon}\beta^2 + d^{4+\varepsilon}\beta)$, since there are $O(d\beta)$ loop iterations for these bases instead of $O(d^2\beta)$ for random bases (see Remark 3, [19]). Moreover, some heuristic results show that when dealing with this kind of bases, and when d, β grow to infinity, one obtains $\Theta(d^3\beta^2)$ when $\beta = \Omega(d^2)$, and $\Theta(d^2\beta^2)$ when β is significantly larger than d (see Heuristic 3, [20]).

Recently, in 2011, Novocin, Stehlé and Villard [22] proposed a new improved LLL-type algorithm that is quasi-linear in β. This led to the name $\tilde{\mathrm{L}}^1$. It is guaranteed to terminate in time $O(d^{5+\varepsilon}\beta + d^{\omega+1+\varepsilon}\beta^{1+\varepsilon})$ for any basis, where ω is a valid exponent from matrix multiplications. To bound ω, we have $2 < \omega \leq 3$. A typical setting in [22] is $\omega = 2.3$.

In [27], van Hoeij and Novocin proposed a gradual sub-lattice reductions algorithm based on LLL that deals with knapsack-type basis. Unlike other LLL-type reduction algorithms, it only produces a basis of a sub-lattice. This algorithm uses a worst-case $O(d^7 + d^3\beta^2)$ time complexity.

For more improvements on LLL with respect to d, we refer readers to [18,25,10,11].

With regard to the quality of a reduced basis for an arbitrary lattice, the following theorem provides an upper bound.

Theorem 2. *For a lattice $\mathcal{L}$, if $(b_1, \ldots, b_n)$ form an LLL-reduced basis of $\mathcal{L}$, then,*

$$\forall i, \|b_i\| \leq 2^{\frac{d-1}{2}} \max(\lambda_i(\mathcal{L})). \tag{2}$$

Therefore, assuming a uniform distribution, we have the following.

1. If a modular knapsack problem follows a uniform distribution, then its corresponding basis forms a random lattice.
2. If $\mathcal{L}$ is a random lattice, then $\lambda_i(\mathcal{L})$ is with respect to Equation 1.
3. From Equation 1 and 2, we have $\|b_i\| \leq 2^{\frac{d-1}{2}}\sqrt{\frac{d}{2\pi e}}\det(\mathcal{L})^{\frac{1}{d}}$ for a random lattice.

Hence, for a modular knapsack-type basis, if $B = (b_1, \ldots, b_d)$ forms its LLL-reduced basis, then

$$\|b_i\| < 2^d \det(\mathcal{L})^{\frac{1}{d}}, \quad 1 \leq i \leq d.$$

In terms of the quality of $\|b_1\|$, the work in [4] shows that on average cases, LLL-type reduction algorithms is able to find a short vector with a Hermite factor 1.0219^d, while on worst cases, 1.0754^d, respectively. Further, heuristically, it is impossible to find vectors with Hermite factor $< 1.01^d$ using LLL-type algorithms [4]. By contrast, a recent work of BKZ 2.0 [2] finds a vector with a Hermite factor as small as 1.0099^d.

It has been shown that other lattice reduction algorithms, for instance, BKZ [24,7], and BKZ 2.0 [2], produce a basis with better quality. However, in general, they are too expensive to use. We also note those methods require to perform at least one LLL reduction.

As for low density knapsack-type basis, the Hermite factor of the basis is large. This implies that, in general, the output basis of most reduction algorithms contains the demanded short vector. In this case, the time complexity is more important, compared with the quality of the reduced basis/vectors. For this reason, in this paper, we focus only on LLL-type reduction algorithms.

3 Our Reduction Methodology

3.1 A Methodology for Lattice Reduction

In this subsection, we do not propose an algorithm for lattice reduction but rather a methodology applicable to *all* lattice reduction algorithms for the knapsack problem with uniform distribution.

Let $\mathcal{A}$ be an LLL-type reduction algorithm that returns an LLL-reduced basis B_{red} of a lattice $\mathcal{L}$ of dimension d, where $B_{red} = (b_1, \dots, b_d)$, $0 < \|b_i\| < c_0^d \det(\mathcal{L})^{\frac{1}{d}}$ for certain constant c_0. The running time will be $c_1 d^{a_1} \beta^{b_1} + c_2 d^{a_2} \beta^{b_2}$, where a_1, b_1, a_2 and b_2 are all parameters, c_1 and c_2 are two constants. Without losing generality, assuming $a_1 \geq a_2$, $b_1 \leq b_2$ (if not, then one term will overwhelm the other, and hence, making the other term negligible). We note that this is a formalization of all LLL-type reduction algorithms.

For a knapsack-type basis B of $\mathcal{L}$, where most of the weight of β are from the first column of the basis matrix $B = (b_1, b_2, \dots, b_d)$, it holds that $2^\beta \sim \det(\mathcal{L})$. Moreover, for any sub-lattice $\mathcal{L}_s$ of $\mathcal{L}$ that is spanned by a subset of row vectors $\{b_1, b_2, \dots, b_d\}$, it is easy to prove that $\det(\mathcal{L}_s) \sim 2^\beta$. In addition, since we assume a uniform distribution, the sub-lattice spanned by the subset of vectors can be seen as a random lattice. Note that the bases of those sub-lattice are knapsack-type basis, so if one needs to ensure the randomness, one is required to add a new vector $\langle X_0, 0, \dots, 0 \rangle$ to the basis and convert it to a modular one. One can verify that this modification will not change the asymptotic complexity. Nevertheless, in practice, it is natural to omit this procedure.

We firstly pre-process the basis, so that the weight is as equally distributed into all columns as possible, and therefore, the maximum norm of the new basis is reduced. Suppose we cut the basis into d/k blocks and each block contains k vectors. Then one applies $\mathcal{A}$ on each block. Since we know that the determinant of each block is $\sim 2^\beta$, this pre-processing gives us a basis with smaller maximum norm $\sim c_0^k 2^{\beta/k}$. Further, since the pre-processed basis and the initial basis span the same lattice, the pre-processing will not affect the quality of reduced basis that a reduction algorithm returns.

Below, we show an example of how this methodology works with dimension 4 knapsack-type basis, where we cut $\mathcal{L}$ into two sub-lattices and pre-process them independently. As a result, $X_i \sim 2^\beta$, while $x_{i,j} \lesssim c_0^2 2^{\frac{\beta}{2}}$ for a classic LLL-type reduction algorithm.

$$B_{before} = \begin{pmatrix} X_1 & 1 & 0 & 0 & 0 \\ X_2 & 0 & 1 & 0 & 0 \\ \hline X_3 & 0 & 0 & 1 & 0 \\ X_4 & 0 & 0 & 0 & 1 \end{pmatrix} \Longrightarrow B_{after} = \begin{pmatrix} x_{1,1} & x_{1,2} & x_{1,3} & 0 & 0 \\ x_{2,1} & x_{2,2} & x_{2,3} & 0 & 0 \\ \hline x_{3,1} & 0 & 0 & x_{3,4} & x_{3,5} \\ x_{4,1} & 0 & 0 & x_{4,4} & x_{4,5} \end{pmatrix}$$

Now we examine the complexity. The total time complexity of this pre-processing is $c_1 dk^{a_1-1}\beta^{b_1} + c_2 dk^{a_2-1}\beta^{b_2}$. The complexity of the final reduction now becomes $c_1 d^{a_1}(k\log_2(c_0) + \beta/k)^{b_1} + c_2 d^{a_2}(k\log_2(c_0) + \beta/k)^{b_2}$. Therefore, as long as

$$\begin{aligned} c_1 d^{a_1}(k\log_2(c_0) + \beta/k)^{b_1} + c_2 d^{a_2}(k\log_2(c_0) + \beta/k)^{b_2} \\ + c_1 dk^{a_1-1}\beta^{b_1} + c_2 dk^{a_2-1}\beta^{b_2} < c_1 d^{a_1}\beta^{b_1} + c_2 d^{a_2}\beta^{b_2}, \end{aligned} \tag{3}$$

conducting the pre-processing will reduce the complexity of whole reduction.

In the case where $k\log_2(c_0)$ is negligible compared with β/k, we obtain:

$$\begin{aligned} c_1 d^{a_1}(\beta/k)^{b_1} + c_2 d^{a_2}(\beta/k)^{b_2} + c_1 dk^{a_1-1}\beta^{b_1} + c_2 dk^{a_2-1}\beta^{b_2} \\ < c_1 d^{a_1}\beta^{b_1} + c_2 d^{a_2}\beta^{b_2}. \end{aligned}$$

Therefore,

$$c_1\left(d^{a_1} - \frac{d^{a_1}}{k^{b_1}} - dk^{a_1-1}\right)\beta^{b_1} + c_2\left(d^{a_2} - \frac{d^{a_2}}{k^{b_2}} - dk^{a_2-1}\right)\beta^{b_2} > 0.$$

Taking L^2 as an example, where $a_1 = 4$, $b_1 = 2$, $a_2 = 5$ and $b_2 = 1$, let $k = d/2$, we obtain $c_1(\frac{7}{8}d^4 - 4d^2)\beta^2 + c_2(\frac{15}{16}d^5 - 2d^4)\beta$ from the left hand side, which is positive for dimension $d > 2$. This indicates that, in theory, when dealing with a knapsack-type basis, one can always achieve a better complexity by cutting the basis into two halves and pre-process them independently. This leads to the recursive reduction in the next section.

3.2 Recursive Reduction with LLL-Type Algorithms

The main idea is to apply our methodology to an input basis recursively, until one arrives to sub-lattice basis with dimension 2. In doing so, we achieve a upper bounded complexity of $O(d^{a_1-b_1}\beta^{b_1} + d^{a_2-b_2}\beta^{b_2})$. For simplicity, we deal with lattice whose dimension equals to a power of 2, although same principle is applicable to lattices with arbitrary dimensions.

Algorithm. We now describe our recursive reduction algorithm with LLL-type reduction algorithms. Let $LLL(\cdot)$ an LLL reduction algorithm that for any lattice basis B, it returns a reduced basis B_r. Algorithm 1 describes our algorithm, where B is a knapsack-type basis of a d-dimensional lattice, and d is a power of 2.

Since we have proven that, for any dimension of knapsack-type basis, it is always better to reduce its sub-lattice in advance as long as Equation 3 holds, it is straightforward to draw the following conclusion: the best complexity to reduce a knapsack-type basis with LLL-type reduction algorithms occurs when one cuts the basis recursively until one arrives with dimension 2 sub-lattices.

Algorithm 1. Recursive Reduction with LLL algorithm

```
Input: $B, d$
Output: $B_r$
  $number\_of\_rounds \leftarrow \log_2 d$
  $B_b \leftarrow B$
  for $i \leftarrow 1 \rightarrow number\_of\_rounds$ do
    $dim\_of\_sublattice \leftarrow 2^i$
    $number\_of\_blocks \leftarrow d/dim\_of\_sublattice$
    $B_r \leftarrow EmptyMatrix()$
    for $j \leftarrow 1 \rightarrow number\_of\_blocks$ do
      $B_t \leftarrow SubMatrix(B_b, (j-1)*dim\_of\_sublattice+1, 1, j*dim\_of\_sublattice, d)$
      $B_t \leftarrow LLL(B_t)$
      $B_r \leftarrow VerticalJoin(B_r, B_t)$
    end for
    $B_b \leftarrow B_r$
  end for
```

In Algorithm 1, $EmptyMatrix(\cdot)$ is to generate a 0 by 0 matrix; $SubMatrix$ (B, a, b, c, d) is to extract a matrix from B, starting from a-th row and b-th column, finishing at c-th row and d-th column; while $VerticalJoin(A, B)$ is to adjoin two matrix with same number of columns vertically.

Complexity. In the following, we prove that the complexity of our algorithm is $O(d^{a_1-b_1}\beta^{b_1} + d^{a_2-b_2}\beta^{b_2})$, assuming $\rho < 1$.

For the i-th round, to reduce a single block takes $c_1 2^{ia_1}(\frac{\beta}{2^{i-1}})^{b_1} + c_2 2^{ia_2}(\frac{\beta}{2^{i-1}})^{b_2}$, while there exist $\frac{d}{2^i}$ such blocks. Hence, the total complexity is as follows:

$$
\begin{aligned}
&\sum_{i=1}^{\log_2 d} \left(\frac{d}{2^i}\right)\left(c_1 2^{ia_1}(\beta/2^{i-1})^{b_1} + c_2 2^{ia_2}(\beta/2^{i-1})^{b_2}\right) \\
&\quad = d \cdot \sum_{i=1}^{\log_2 d} \left(c_1 2^{i(a_1-b_1-1)+b_1}\beta^{b_1} + c_2 2^{i(a_2-b_2-1)+b_2}\beta^{b_2}\right) \\
&\quad = c_1 2^{b_1} d\beta^{b_1} \left(\sum_{i=1}^{\log_2 d} 2^{(a_1-b_1-1)i}\right) + c_2 2^{b_2} d\beta^{b_2} \left(\sum_{i=1}^{\log_2 d} 2^{(a_2-b_2-1)i}\right) \\
&\quad < c_1 2^{b_1} d\beta^{b_1} \left(2^{(\log_2 d+1)(a_1-b_1-1)}\right) + c_2 2^{b_2} d\beta^{b_2} \left(2^{(\log_2 d+1)(a_2-b_2-1)}\right) \\
&\quad < c_1 2^{b_1} d\beta^{b_1} (2d)^{a_1-b_1-1} + c_2 2^{b_2} d\beta^{b_2} (2d)^{a_2-b_2-1} \\
&\quad < c_1 2^{a_1-1} d^{a_1-b_1}\beta^{b_1} + c_2 2^{a_2-1} d^{a_2-b_2}\beta^{b_2}.
\end{aligned}
$$

As a result, we obtain a new time complexity $O(d^{a_1-b_1}\beta^{b_1} + d^{a_2-b_2}\beta^{b_2})$.

4 Applying Recursive Reduction to L^2

We adapt the classic L^2 as an example. The L^2 algorithm uses a worst-case complexity of $c_1 d^4 \beta^2 + c_2 d^5 \beta$ for arbitrary basis. Therefore, applying our recursive methodology, one obtains

$$\begin{aligned}
\sum_{i=1}^{\log_2 d} \left(\frac{d}{2^i}\right) \left(c_1 2^{4i} \left(\frac{\beta}{2^{i-1}}\right)^2 + c_2 2^{5i} \left(\frac{\beta}{2^{i-1}}\right)\right)
&= \sum_{i=1}^{\log_2 d} (4c_1 d 2^i \beta^2 + 2c_2 d 2^{3i} \beta) \\
&= 4c_1 d\beta^2 \left(\sum_{i=1}^{\log_2 d} 2^i\right) + 2c_2 d\beta \left(\sum_{i=1}^{\log_2 d} 2^{3i}\right) \\
&< 4c_1 d\beta^2 (2d) + 2c_2 d\beta 1.15 d^3 \\
&< 8c_1 d^2 \beta^2 + 2.3 c_2 d^4 \beta.
\end{aligned}$$

Now we compare our complexity with the original L^2 algorithm. As mentioned earlier, when applying to a knapsack-type basis, the provable worst-case complexity of L^2 becomes $c_1 d^3 \beta^2 + c_2 d^4 \beta$ rather than $c_1 d^4 \beta^2 + c_2 d^5 \beta$ as for a random basis. However, it is worth pointing out that in practice, one can achieve a much better result than a worst case, since the weight of most X_i is equally distributed into all the columns. Heuristically, one can expect $\Theta(c_1 d^2 \beta^2)$ when d, β go to infinity and $\beta \gg d$.

Input a knapsack-type basis, the L^2 algorithm (and almost all other LLL-type reduction algorithms) tries to reduce the first k rows, then the $k+1$ row, $k+2$ row, etc. For a given $k+1$ step, the current basis has the following shape:

$$B_{knap-L^2} = \begin{pmatrix}
x_{1,1} & x_{1,2} & \dots & x_{1,k+1} & 0 & 0 & \dots & 0 \\
x_{2,1} & x_{2,2} & \dots & x_{2,k+1} & 0 & 0 & \dots & 0 \\
\vdots & \vdots & \dots & \vdots & \vdots & \vdots & \dots & \vdots \\
x_{k,1} & x_{k,2} & \dots & x_{k,k+1} & 0 & 0 & \dots & 0 \\
X_{k+1} & 0 & \dots & 0 & 1 & 0 & \dots & 0 \\
X_{k+2} & 0 & \dots & 0 & 0 & 1 & \dots & 0 \\
\vdots & \vdots & \dots & \vdots & \vdots & \vdots & \dots & \vdots \\
X_d & 0 & \dots & 0 & 0 & 0 & \dots & 1
\end{pmatrix}$$

L^2 will reduce the first $k+1$ rows during this step. Despite that most of the entries are with small elements ($\|x_{i,j}\| \sim O(2^{\frac{\beta}{k}})$), the worse-case complexity of current step still depends on the last row of current step, i.e., $\langle X_{k+1}, 0, \dots, 0, 1, 0, \dots, 0\rangle$.

For the recursive reduction, on the final step, the input basis is in the form of:

$$B_{rec-L^2} = \begin{pmatrix} x_{1,1} & x_{1,2} \dots x_{1,\frac{d}{2}+1} & 0 & 0 & \dots & 0 \\ x_{2,1} & x_{2,2} \dots x_{2,\frac{d}{2}+1} & 0 & 0 & \dots & 0 \\ \vdots & \vdots \dots \vdots & \vdots & \vdots & \dots & \vdots \\ x_{\frac{d}{2},1} & x_{k,2} \dots x_{\frac{d}{2},\frac{d}{2}+1} & 0 & 0 & \dots & 0 \\ x_{\frac{d}{2}+1,1} & 0 \dots 0 & x_{\frac{d}{2}+1,\frac{d}{2}+2} & x_{\frac{d}{2}+1,\frac{d}{2}+3} & \dots & x_{\frac{d}{2}+1,d+1} \\ x_{\frac{d}{2}+2,1} & 0 \dots 0 & x_{\frac{d}{2}+2,\frac{d}{2}+2} & x_{\frac{d}{2}+2,\frac{d}{2}+3} & \dots & x_{\frac{d}{2}+2,d+1} \\ \vdots & \vdots \dots \vdots & \vdots & \vdots & \dots & \vdots \\ x_{d,1} & 0 \dots 0 & x_{d,\frac{d}{2}+2} & x_{d,\frac{d}{2}+3} & \dots & x_{d,d+1} \end{pmatrix}$$

Note that the weight of X_i is equally distributed into $\frac{d}{2}+1$ columns. Hence, the bit length of maximum norm of basis is reduced from β to approximately $d \log_2 c_0 + 2\beta/d$. Therefore, we achieve a better time complexity. In fact, the provable new complexity is of the same level of the heuristic results observed in practice, when $\beta \gg d$.

5 Special Case: Principal Ideal Lattice Basis

In this section, we present a technique when dealing with a principal ideal lattice basis. Due to the special form of a principal ideal lattice, we are able to reduce the number of reductions in each round to 1, with a cost of $O(d)$ additional vectors for the next round. This technique does not effect the asymptotic complexity, however, in practice, it will accelerate the reduction.

$$B_I = \begin{pmatrix} \delta & 0\,0 \dots 0 \\ -\alpha \bmod \delta & 1\,0 \dots 0 \\ -\alpha^2 \bmod \delta & 0\,1 \dots 0 \\ \vdots & \vdots\ \vdots \ddots \vdots \\ -\alpha^{d-1} \bmod \delta & 0\,0 \dots 1 \end{pmatrix} \Longrightarrow B_I' = \begin{pmatrix} \delta & 0 & 0 \dots & 0 & 0 \\ -\alpha & 1 & 0 \dots & 0 & 0 \\ 0 & -\alpha & 1 \dots & 0 & 0 \\ \vdots & \vdots & \vdots \ddots & \vdots & \vdots \\ 0 & 0 & 0 \dots & -\alpha & 1 \end{pmatrix}.$$

Let $X_0 = \delta$, one obtains B_I in the above form. From B_I, one constructs a new basis B_I'. Then, one can obtain a generator matrix of $\mathcal{L}(B_I)$ by inserting some vectors in $\mathcal{L}$ to B_I'.

The following example shows how to construct G with $d = 5$. In this example, since vector $\langle 0, 0, \delta, 0, 0\rangle$ is a valid vector in $\mathcal{L}(B)$, B and G span a same lattice. Applying a lattice reduction algorithm over G will return a matrix with the top row that is a zero vector, while the rest form a reduced basis of $\mathcal{L}$.

$$G = \left(\begin{array}{ccccc} \delta & 0 & 0 & 0 & 0 \\ -\alpha & 1 & 0 & 0 & 0 \\ 0 & -\alpha & 1 & 0 & 0 \\ \hline 0 & 0 & \delta & 0 & 0 \\ 0 & 0 & -\alpha & 1 & 0 \\ 0 & 0 & 0 & -\alpha & 1 \end{array}\right) \Longrightarrow \left(\begin{array}{ccccc} x_{1,1} & x_{1,2} & x_{1,3} & 0 & 0 \\ x_{2,1} & x_{2,2} & x_{2,3} & 0 & 0 \\ x_{3,1} & x_{3,2} & x_{3,3} & 0 & 0 \\ \hline 0 & 0 & x_{1,1} & x_{1,2} & x_{1,3} \\ 0 & 0 & x_{2,1} & x_{2,2} & x_{2,3} \\ 0 & 0 & x_{3,1} & x_{3,2} & x_{3,3} \end{array}\right)$$

To reduce G, we adopt our recursive reduction methodology. We firstly reduce the top half of G. Since the second half is identical to the top half, except the position of the elements, we do not need to reduce the second half. Indeed, we use the result of the top half block and then shift all the elements. In this case, during our recursive reduction, for round i, instead of doing $d/2^i$ reductions, one need to perform only one reduction. Finally, one reduces the final matrix G, removes all the zero vectors and start a new round.

With our technique, the number of vectors grows, and this may increase the complexity of the next round. For the i-th round, the number of vectors grows by $d/2^{i-1} - 1$. It will be negligible when $d/2^i \ll d$. For instance, if we adopt this approach between the second last round and the last round, this approach will only increase the number of vectors by 1, while if one uses it prior to the first round, the number of rows will almost be doubled. In practice, one can choose to adopt this technique for each round only when it accelerates the reduction.

We note that the asymptotic complexity remains the same, since generally speaking, the number of vectors remains $O(d)$ as before, while the asymptotic complexity concerns only d and β.

6 Conclusion

In this paper, we presented a methodology for lattice reduction algorithms used for solving low density modular knapsack problems. The complexity of polynomial time lattice reduction algorithms relies on the dimension d and the bit length β of maximum norm of input basis. We prove that for a knapsack-type basis, it is always better to pre-process the basis by distributing the weight to many columns as equally as possible. Using this methodology recursively, we are able to reduce β to approximately $2\beta/d$, and consequently, we successfully reduce the entire complexity.

We then demonstrated our technique over the floating-point LLL algorithm. We obtain a provable upper bounded complexity of $O(d^{2+\varepsilon}\beta^2 + d^{4+\varepsilon}\beta)$, which is by far the best provable time complexity for a knapsack-type basis.

References

1. Ajtai, M.: The shortest vector problem in l_2 is NP-hard for randomized reductions (extended abstract). In: Thirtieth Annual ACM Symposium on the Theory of Computing (STOC 1998), pp. 10–19 (1998)
2. Chen, Y., Nguyên, P.Q.: BKZ 2.0: Better Lattice Security Estimates. In: Lee, D.H. (ed.) ASIACRYPT 2011. LNCS, vol. 7073, pp. 1–20. Springer, Heidelberg (2011)
3. Coster, M.J., Joux, A., LaMacchia, B.A., Odlyzko, A.M., Schnorr, C.-P., Stern, J.: Improved low-density subset sum algorithms. Computational Complexity 2, 111–128 (1992)
4. Gama, N., Nguyên, P.Q.: Predicting Lattice Reduction. In: Smart, N.P. (ed.) EUROCRYPT 2008. LNCS, vol. 4965, pp. 31–51. Springer, Heidelberg (2008)
5. Gentry, C.: Fully homomorphic encryption using ideal lattices. In: Mitzenmacher, M. (ed.) STOC, pp. 169–178. ACM (2009)

6. Goldstein, D., Mayer, A.: On the equidistribution of Hecke points. Forum Mathematicum 15, 165–189 (2006)
7. Hanrot, G., Pujol, X., Stehlé, D.: Analyzing Blockwise Lattice Algorithms Using Dynamical Systems. In: Rogaway, P. (ed.) CRYPTO 2011. LNCS, vol. 6841, pp. 447–464. Springer, Heidelberg (2011)
8. Kannan, R.: Improved algorithms for integer programming and related lattice problems. In: Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing, STOC 1983, pp. 193–206. ACM, New York (1983)
9. Karp, R.M.: Reducibility among combinatorial problems. In: Miller, R.E., Thatcher, J.W. (eds.) Complexity of Computer Computations. The IBM Research Symposia Series, pp. 85–103. Plenum Press, New York (1972)
10. Koy, H., Schnorr, C.-P.: Segment LLL-Reduction of Lattice Bases. In: Silverman, J.H. (ed.) CaLC 2001. LNCS, vol. 2146, pp. 67–80. Springer, Heidelberg (2001)
11. Koy, H., Schnorr, C.-P.: Segment LLL-Reduction with Floating Point Orthogonalization. In: Silverman, J.H. (ed.) CaLC 2001. LNCS, vol. 2146, pp. 81–96. Springer, Heidelberg (2001)
12. Lagarias, J.C., Odlyzko, A.M.: Solving low-density subset sum problems. J. ACM 32(1), 229–246 (1985)
13. Lai, M.K.: Knapsack cryptosystems: The past and the future (2001)
14. Lenstra, A.K., Lenstra, H.W., Lovász, L.: Factoring polynomials with rational coefficients. Mathematische Annalen 261, 513–534 (1982)
15. Lovász, L.: An Algorithmic Theory of Numbers, Graphs and Convexity. In: CBMS-NSF Regional Conference Series in Applied Mathematics, vol. 50. SIAM Publications (1986)
16. Micciancio, D., Goldwasser, S.: Complexity of Lattice Problems, A Cryptographic Perspective. Kluwer Academic Publishers (2002)
17. Minkowski, H.: Geometrie der Zahlen. B. G. Teubner, Leipzig (1896)
18. Morel, I., Stehlé, D., Villard, G.: H-LLL: using householder inside LLL. In: ISSAC, pp. 271–278 (2009)
19. Nguyên, P.Q., Stehlé, D.: Floating-Point LLL Revisited. In: Cramer, R. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 215–233. Springer, Heidelberg (2005)
20. Nguyên, P.Q., Stehlé, D.: LLL on the Average. In: Hess, F., Pauli, S., Pohst, M. (eds.) ANTS 2006. LNCS, vol. 4076, pp. 238–256. Springer, Heidelberg (2006)
21. Nguyên, P.Q., Stern, J.: Adapting Density Attacks to Low-Weight Knapsacks. In: Roy, B. (ed.) ASIACRYPT 2005. LNCS, vol. 3788, pp. 41–58. Springer, Heidelberg (2005)
22. Novocin, A., Stehlé, D., Villard, G.: An LLL-reduction algorithm with quasi-linear time complexity: extended abstract. In: Fortnow, L., Vadhan, S.P. (eds.) STOC, pp. 403–412. ACM (2011)
23. Pujol, X., Stehlé, D., Cade, D.: fplll library, http://perso.ens-lyon.fr/xavier.pujol/fplll/
24. Schnorr, C.-P.: A more efficient algorithm for lattice basis reduction. J. Algorithms 9(1), 47–62 (1988)
25. Schnorr, C.-P.: Fast LLL-type lattice reduction. Inf. Comput. 204(1), 1–25 (2006)
26. Smart, N.P., Vercauteren, F.: Fully Homomorphic Encryption with Relatively Small Key and Ciphertext Sizes. In: Nguyên, P.Q., Pointcheval, D. (eds.) PKC 2010. LNCS, vol. 6056, pp. 420–443. Springer, Heidelberg (2010)
27. van Hoeij, M., Novocin, A.: Gradual sub-lattice reduction and a new complexity for factoring polynomials. Algorithmica 63(3), 616–633 (2012)

The Boomerang Attacks on the Round-Reduced Skein-512*

Hongbo Yu[1], Jiazhe Chen[3,4], and Xiaoyun Wang[2,3]

[1] Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China
[2] Institute for Advanced Study, Tsinghua University, Beijing 100084, China
{yuhongbo,xiaoyunwang}@mail.tsinghua.edu.cn
[3] Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, School of Mathematics, Shandong University, China
[4] KU Leuven, ESAT/COSIC and IBBT, Belgium
jiazhechen@mail.sdu.edu.cn

Abstract. The hash function Skein is one of the five finalists of the NIST SHA-3 competition. It is based on the block cipher Threefish which only uses three primitive operations: modular addition, rotation and bitwise XOR (ARX). This paper studies the boomerang attacks on Skein-512. Boomerang distinguishers on the compression function reduced to 32 and 36 rounds are proposed, with time complexities $2^{104.5}$ and 2^{454} hash computations respectively. Examples of the distinguishers on 28 and 31 rounds are also given. In addition, the boomerang distinguishers are applicable to the key-recovery attacks on reduced Threefish-512. The time complexities for key-recovery attacks reduced to 32-/33-/34-round are about 2^{181}, 2^{305} and 2^{424} encryptions. Because the previous boomerang distinguishers for Threefish-512 are in fact not compatible [14], our attacks are the first valid boomerang attacks for the reduced-round Skein-512.

Keywords: Hash function, Boomerang attack, Threefish, Skein.

1 Introduction

Cryptographic hash functions, which provide integrity, authentication etc., are very important in modern cryptology. In 2007, NIST launched a hash competition for a new hash standard (SHA-3) as the most widely used hash functions MD5 and SHA-1 were broken [19,20]. Now the competition has come into the third round (the final round), and 5 finalists out of the candidates are selected. The finalist Skein [7] is a ARX-type hash function (based on modular addition, rotation and exclusive-OR). The core of the compression function of Skein is a tweakable block cipher called Threefish, which is proposed with 256-, 512-, 1024-bit block sizes and 72, 72, 80 rounds, respectively. When the algorithm entered

* Supported by 973 program (No.2013CB834205), the National Natural Science Foundation of China (No. 61133013), and the Tsinghua University Initiative Scientific Research Program (No.20111080970).

into the second round, the authors changed the rotation constants to refine the algorithm, and after it was selected as a finalist, the constants used in the key schedule were updated to resist the rotational attack [10,11].

During the competition, Skein has been attracting the attentions of the cryptanalysts, and there are several cryptanalytic results on the security of the compression function of Skein and its based block cipher Threefish. At Asiacrypt 2009 [1], Aumasson *et al.* used the boomerang attack to launch a key recovery attack on Threefish-512 reduced to 32 rounds and the known-key distinguisher to 35 rounds under the old rotation constants. However, we find that their differential paths used in the boomerang attacks employ an inverse permutation instead of the original one. In 2010, Chen *et al.* also proposed a boomerang attack for the key recovery of Threefish-512 reduced to 33 and 34 rounds on the new rotation constants using the method of modular differential. Recently Leurent *et al.* [14] gave a boomerang distinguisher for 32-round compression function of Skein-256, and they also pointed that the differential paths in [6] are incompatible. We correct the paths in [1] with the right permutation and show that they are also incompatible under the old rotation constants due to similar contradictions as in [6]. Besides the boomerang attacks, some other attack methods also appeared for Skein. At CANS 2010 [16], Su *et al.* presented free-start near-collisions of Skein-256/-512 compression functions reduced to 20 rounds and Skein-1024 reduced to 24 rounds. At Asiacrypt 2010 [11], Khovratovich *et al.* combined the rotational attack and the rebound attack, and gave distinguishers on 53-round Skein-256 and 57-round Skein-512 respectively, and their technique depends on the constants used in the key schedule. In paper [21], Yu *et al.* gave a near-collision attack for Skein-256 using the rebound attack which was also been shown using incompatible paths [15]. In [12], Khovratovich *et al.* also gave a preimage attack on 22-round Skein-512 hash function and 37-round Skein-512 compression function by the biclique method.

Our Contribution. In this paper, we study the boomerang distinguishers on round-reduced Skein-512. Our analysis is based on two related-key differential paths of Threefish-512 with high probability. In order to solve the incompatibility pointed out in [14], we select differences for the key words and tweaks on the 59-*th* bit instead of the 64-*th* bit (the 64-*th* is the most significant bit) for the top path.

We also reveal that the four paths in the middle 8 rounds are not independent, the probability of the distinguisher in the middle 8 rounds is much higher than the average probability. Based on the differential paths, we give boomerang distinguisher on the compression function of Skein-512 reduced to 32 round with complexity $2^{104.5}$. The distinguisher can be extended to 36 rounds by adding two more rounds on the top and bottom of the differential paths respectively. Our boomerang distinguishers are applicable to the related-key key-recovery attacks on Threefish-512 reduced to 32, 33 and 34 rounds for 1/4 of the keys. Table 1 summarizes our results.

The rest of the paper is organized as follows. In Sect.2, we give a brief description of Skein-512. Sect.3 summaries the boomerang attack. Sect.4 leverages the boomerang technique to the compression functions of Skein-512. In Sec.5, we introduce the key-recovery attacks based on our boomerang distinguishers. Finally, a conclusion of the paper is given in Sect.6.

Table 1. Summary of the attacks on Skein (only the attacks independent of the constants are mentioned)

Attack	CF/KP	Rounds	Time	Ref.
Near collisions(Skein-256)	CF	20	2^{60}	[16]
Near Collisions(Skein-256)	CF	32	2^{105}	[21]*
Pseudo-preimage(Skein-512)	CF	37	$2^{511.2}$	[12]
Boomerang Dist.(Skein-256)	CF	28	2^{24}	[14]
Boomerang Dist.(Skein-256)	KP	32	2^{57}	
Boomerang Dist.(Skein-256)	CF	32	2^{114}	
Key Recovery (Threefish-512)	KP	32	2^{312}	[1]*
Boomerang Dist. (Threefish-512)	KP	35	2^{478}	
Key Recovery (Threefish-512)	KP	32	2^{189}	[6]*
Key Recovery (Threefish-512)	KP	33	$2^{324.6}$	
Key Recovery (Threefish-512)	KP	34	$2^{474.4}$	
Boomerang Dist.(Skein-512)	CF	28	$2^{40.5}$	Sec.4
Boomerang Dist.(Skein-512)	CF	31	2^{32}†	
Boomerang Dist.(Skein-512)	CF	32	$2^{56.5}$†	
Boomerang Dist.(Skein-512)	CF	32	$2^{104.5}$	
Boomerang Dist.(Skein-512)	CF	33	2^{125}†	
Boomerang Dist.(Skein-512)	CP	34	$2^{190.6}$†	
Boomerang Dist.(Skein-512)	CP	35	2^{308}†	
Boomerang Dist.(Skein-512)	CP	36	2^{454}†	
Key-recovery (Threefish-512)	KP	32	2^{181}	Sec.5
Key-recovery (Threefish-512)	KP	33	2^{305}	
Key-recovery (Threefish-512)	KP	34	2^{424}	

KP: Keyed permutation, CF: Compression Function.
*:The differential paths are incompatible.
†: The initial and final subkeys are not included.

2 Description of Skein-512

Skein is designed by Ferguson *et al.*, which is one of the SHA-3 finalists. It supports three different internal state sizes (256, 512, and 1024 bits) and each of these state sizes can support any output size. The word size which Skein operates on is 64 bits. Skein is based on the UBI (Unique Block Iteration) chaining mode that uses block cipher Threefish to build a compression function.

The compression function of Skein can be defined as $H = E(K, T, M) \oplus M$, where $E(K, T, M)$ is the block cipher Threefish, M is the plaintext, K is the master key and T is the tweak value. For Skein-512, both M and K are 512 bits, and the length of T is 128 bits. Let us denote $V_i = (a_i, b_i, c_i, d_i, e_i, f_i, g_i, h_i)$ as the output value of the i-th round, where a_i, b_i, ..., h_i are 64-bit words. Let $V_0 = M$ be the plaintext, the encryption procedure of Threefish-512 is carried out for $i = 1$ to 72 as follows.

If $(i-1) \mod 4 = 0$, first compute

$$\begin{gathered}
\hat{a}_{i-1} = a_{i-1} + K_{(i-1)/4,a},\ \hat{b}_{i-1} = b_{i-1} + K_{(i-1)/4,b},\\
\hat{c}_{i-1} = c_{i-1} + K_{(i-1)/4,c},\ \hat{d}_{i-1} = d_{i-1} + K_{(i-1)/4,d},\\
\hat{e}_{i-1} = e_{i-1} + K_{(i-1)/4,e},\ \hat{f}_{i-1} = f_{i-1} + K_{(i-1)/4,f},\\
\hat{g}_{i-1} = g_{i-1} + K_{(i-1)/4,g},\ \hat{h}_{i-1} = h_{i-1} + K_{(i-1)/4,h},
\end{gathered}$$

where $K_{(i-1)/4,a}$, $K_{(i-1)/4,b}$, ..., $K_{(i-1)/4,h}$ are round subkeys which are involved in every four rounds. Then carry out:

$$\begin{gathered}
a_i = \hat{c}_{i-1} + \hat{d}_{i-1},\ h_i = a_i \oplus (\hat{d}_{i-1} \lll R_{i,1}),\\
c_i = \hat{e}_{i-1} + \hat{f}_{i-1},\ f_i = c_i \oplus (\hat{f}_{i-1} \lll R_{i,2}),\\
e_i = \hat{g}_{i-1} + \hat{h}_{i-1},\ d_i = e_i \oplus (\hat{h}_{i-1} \lll R_{i,3}),\\
g_i = \hat{a}_{i-1} + \hat{b}_{i-1},\ b_i = g_i \oplus (\hat{b}_{i-1} \lll R_{i,0}),
\end{gathered}$$

where $R_{i,1}$ and $R_{i,2}$ are rotation constants which can be found in [7]. For the sake of convenience, we denote $\hat{V}_{i-1} = (\hat{a}_{i-1}, \hat{b}_{i-1}, \hat{c}_{i-1}, \hat{d}_{i-1}, \hat{e}_{i-1}, \hat{f}_{i-1}, \hat{g}_{i-1}, \hat{h}_{i-1})$.

If $(i-1) \mod 4 \neq 0$, compute

$$\begin{gathered}
a_i = c_{i-1} + d_{i-1},\ h_i = a_i \oplus (d_{i-1} \lll R_{i,1}),\\
c_i = e_{i-1} + f_{i-1},\ f_i = c_i \oplus (f_{i-1} \lll R_{i,2}),\\
e_i = g_{i-1} + h_{i-1},\ d_i = e_i \oplus (h_{i-1} \lll R_{i,3}),\\
g_i = a_{i-1} + b_{i-1},\ b_i = g_i \oplus (b_{i-1} \lll R_{i,0}).
\end{gathered}$$

After the last round, the ciphertext is computed as $\hat{V}_{72} = (\hat{a}_{72}, \hat{b}_{72}, ..., \hat{h}_{72})$.

The key schedule starts with the master key $K = (k_0, k_1, k_2, k_3, k_4, k_5, k_6, k_7)$ and the tweak value $T = (t_0, t_1)$. First we compute

$$k_8 := 0x1bd11bdaa9fc1a22 \oplus \bigoplus_{i=0}^{7} k_i \qquad \text{and} \qquad t_2 := t_0 \oplus t_1.$$

Then the subkeys are derived for $s = 0$ to 18:

$$\begin{aligned}
K_{s,a} &:= k_{(s+0) \bmod 9}\\
K_{s,b} &:= k_{(s+1) \bmod 9}\\
K_{s,c} &:= k_{(s+2) \bmod 9}\\
K_{s,d} &:= k_{(s+3) \bmod 9}\\
K_{s,e} &:= k_{(s+4) \bmod 9}\\
K_{s,f} &:= k_{(s+5) \bmod 9} + t_{s \bmod 3}\\
K_{s,g} &:= k_{(s+6) \bmod 9} + t_{(s+1) \bmod 3}\\
K_{s,h} &:= k_{(s+7) \bmod 9} + s
\end{aligned}$$

3 The Boomerang Attack

The boomerang attack was introduced by Wagner [17] and first applied to block ciphers; it is an adaptive chosen plaintext and ciphertext attack. Later it was further developed by Kelsey et al. into a chosen plaintext attack called the amplified boomerang attack [13], then Biham *et al.* further developed it into the rectangle attack [3]. The basic idea of the boomerang attack is joining two short differential paths with high probabilities in a quartet. The related-key boomerang attack is proposed in [4] and it uses the related-key differentials instead of the single-key differentials. Let E be a block cipher with block size n bits, and it can be decomposed into two sub-ciphers: $E = E_1 \circ E_0$. For the sub-cipher E_0, there is a differential path $(\alpha, \alpha_k) \to \beta$ with probability p. And for the sub-cipher E_1, there is a differential path $(\gamma, \gamma_k) \to \delta$ with probability q. Then the related-key boomerang attack can be constructed:

- Randomly choose a pair of plaintexts (P_1, P_2) such that $P_2 - P_1 = \alpha$.
- Compute $\mathcal{K}_2 = \mathcal{K}_1 + \alpha_k$, $\mathcal{K}_3 = \mathcal{K}_1 + \gamma_k$ and $\mathcal{K}_4 = \mathcal{K}_1 + \alpha_k + \gamma_k$. Encrypt P_1, P_2 with the related keys $\mathcal{K}_1$ and $\mathcal{K}_2$ to get $C_1 = E_{\mathcal{K}_1}(P_1)$, $C_2 = E_{\mathcal{K}_2}(P_2)$.
- Compute $C_3 = C_1 + \delta$, $C_4 = C_2 + \delta$. Decrypt C_3, C_4 with the related keys $\mathcal{K}_3$ and $\mathcal{K}_4$ to get $P_3 = E^{-1}_{\mathcal{K}_3}(C_3)$, $P_4 = E^{-1}_{\mathcal{K}_4}(C_4)$.
- Check whether $P_4 - P_3 = \alpha$.

It is known that for an n-bit random permutation, $P_4 - P_3 = \alpha$ with probability 2^{-n}. Therefore, the attack is valid if $p^2q^2 > 2^{-n}$.

In the known-key setting, a (related-key) boomerang attack can be used to distinguish a given permutation from a random oracle; it is called known-related-key boomerang attack in [5]. Applying the known-related-key boomerang attack to the compression function in the MMO mode, i.e, $CF(K, M) = E_K(M) + M$, it is possible to start from the middle rounds because the message M and the key K can be selected randomly (refer to [5] and [14]). The (known-related-key) boomerang attack is particularly efficient for the ARX-type hash functions because their compression functions have strong diffusion after several steps, only short differential paths with high probabilities can be found. See Fig. 1 for the schematic view of the boomerang distinguisher for hash functions. The known-related-key boomerang attack for a permutation (or a compression function in the MMO structure) can be summarized as follows.

- Choose a random value X_1 and $\mathcal{K}_1$, compute $X_2 = X_1 + \beta$, $X_3 = X_1 + \gamma$, $X_4 = X_3 + \beta$ and $\mathcal{K}_2 = \mathcal{K}_1 + \beta_k$, $\mathcal{K}_3 = \mathcal{K}_1 + \gamma_k$, $\mathcal{K}_4 = \mathcal{K}_3 + \beta_k$.
- Compute backward from quartets $(X_i, \mathcal{K}_i)_{i=1}^4$ using E_0^{-1} to obtain P_1, P_2, P_3 and P_4.
- Compute forward from quartets $(X_i, \mathcal{K}_i)_{i=1}^4$ using E_1 to obtain C_1, C_2, C_3 and C_4.
- Check whether $P_2 - P_1 = P_4 - P_3 = \alpha$ and $C_3 - C_1 = C_4 - C_2 = \delta$ are fulfilled.

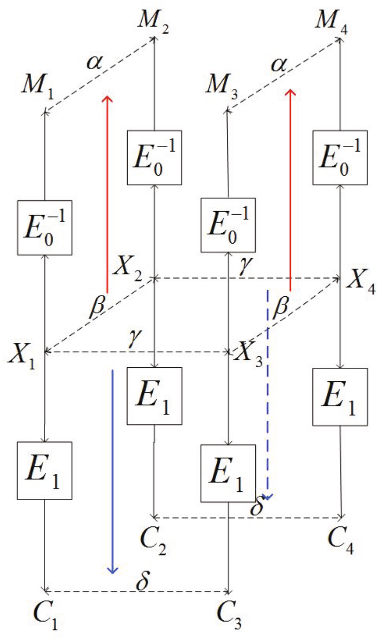


Fig. 1. The boomerang attack

Summary up the previous work [5,14], the boomerang distinguisher falls into three types according to the input and output differences for an n-bit fixed permutation.

- Type I: A quartet satisfies $P_2 - P_1 = P_4 - P_3 = \alpha$ and $C_3 - C_1 = C_4 - C_2 = \delta$ for fixed α and δ. In this case, the generic complexity is 2^n.
- Type II: Only $C_3 - C_1 = C_4 - C_2$ are required (the property is also called zero-sum or second-order differential collision). In this case, the complexity for obtaining such a quartet is $2^{n/3}$ using Wagner's generalized birthday attack [18].
- Type III: A quartet satisfies $P_2 - P_1 = P_4 - P_3$ and $C_3 - C_1 = C_4 - C_2$. In this case, the best known attack still takes time $2^{n/2}$.

4 The Boomerang Distinguisher on Reduced Skein-512

In this section, we describe the known-related-key boomerang attack on Skein-512 reduced to 36 rounds. As mentioned above, the basic idea of our attack is to connect two short differential paths in a quartet. The first step of our attack is to find two short differentials with high probabilities so that the switch in the middle does not contain any contradictions. Secondly, we derive the sufficient conditions for the rounds in the middle, and compute the precise probability of each condition. Thirdly, we correct the conditions in the intermediate rounds by modifying the chaining variables, the key K and the tweak value T. Finally, after the message modification, we search the right quartet that pass the verification of the distinguisher.

4.1 Round-Reduced Differential Paths for Skein-512

The differences of the master key $K = (k_i)_{i=0}^{7}$ and tweak value $T = (t_0, t_1)$ selected for the top differential path are $\Delta k_0 = 0x0400000000000000$, $\Delta t_0 = 0x0400000000000000$ and $\Delta t_1 = 0x0400000000000000$. Suppose $k_{8,59} = t_{0,59} \oplus 1$ and $k_{0,59} = t_{1,59} \oplus 1$, then there is no difference in the fourth subkey. For the bottom path, the MSB differences are set in k_3, k_4 and t_1, and this gives no difference in the eighth subkey. According to the key schedule, the differences for the subkeys $K_i = (K_{i,a}, K_{i,b}, K_{i,c}, K_{i,d}, K_{i,e}, K_{i,f}, K_{i,g}, K_{i,h})$ $(0 \le i \le 9)$ are shown in Tables 2 and 3.

Table 2. The subkey differences of the top path

s	d	$K_{i,a}$	$K_{i,b}$	$K_{i,c}$	$K_{i,d}$	$K_{i,e}$	$K_{i,f}$	$K_{i,g}$	$K_{i,h}$
		Differences							
0	0	k_0	k_1	k_2	k_3	k_4	$k_5 + t_0$	$k_6 + t_1$	$k_7 + 0$
		$\pm 2^{58}$	0	0	0	0	$\pm 2^{58}$	$\pm 2^{58}$	0
1	4	k_1	k_2	k_3	k_4	k_5	$k_6 + t_1$	$k_7 + t_2$	$k_8 + 1$
		0	0	0	0	0	$\pm 2^{58}$	0	$\pm 2^{58}$
2	8	k_2	k_3	k_4	k_5	k_6	$k_7 + t_2$	$k_8 + t_0$	$k_0 + 2$
		0	0	0	0	0	0	0	$\pm 2^{58}$
3	12	k_3	k_4	k_5	k_6	k_7	$k_8 + t_0$	$k_0 + t_1$	$k_1 + 3$
		0	0	0	0	0	0	0	0
4	16	k_4	k_5	k_6	k_7	k_8	$k_0 + t_1$	$k_1 + t_2$	$k_2 + 4$
		0	0	0	0	$\pm 2^{58}$	0	0	0

Table 3. The subkey differences of the bottom path

s	d	$K_{i,a}$	$K_{i,b}$	$K_{i,c}$	$K_{i,d}$	$K_{i,e}$	$K_{i,f}$	$K_{i,g}$	$K_{i,h}$
		Differences							
5	20	k_5	k_6	k_7	k_8	k_0	$k_1 + t_2$	$k_2 + t_0$	$k_3 + 5$
		0	0	0	0	0	2^{63}	0	2^{63}
6	24	k_6	k_7	k_8	k_0	k_1	$k_2 + t_0$	$k_3 + t_1$	$k_4 + 6$
		0	0	0	0	0	0	0	2^{63}
7	28	k_7	k_8	k_0	k_1	k_2	$k_3 + t_1$	$k_4 + t_2$	$k_5 + 7$
		0	0	0	0	0	0	0	0
8	32	k_8	k_0	k_1	k_2	k_3	$k_4 + t_2$	$k_5 + t_0$	$k_6 + 8$
		0	0	0	0	2^{63}	0	0	0
9	36	k_0	k_1	k_2	k_3	k_4	$k_5 + t_0$	$k_6 + t_1$	$k_7 + 9$
		0	0	0	2^{63}	2^{63}	0	2^{63}	0

Table 4. The top differential path used for boomerang attacks of Skein-512

Rd	Shifts	Difference	Pr
2	17, 49 36, 39	0c030025814280b4 08020024800290a0 84689060080a4234 80209020280a0224 603a002310842201 4038002312046020 09421184e3408c32 906008062408c22	2^{-73}
3	44, 9 54, 56	0448004020004010 0448000420000010 2002000002804221 2002000002004021 0044110481000010 0044020401004010 0401000101401014 0001000100401004	2^{-35}
4		0000000000800240 0001000080000200 0000110080004000 0000010000004000 0400000001000010 0400000001000010 0000004400004000 0400000400004000	2^{-24}
K_1		0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0400000000000000 0000000000000000 0400000000000000	–
4	39, 30 34, 24	– 0001000080000200 – 0000010000004000 – 0000000001000010 – 0000000400004000	1
5	13, 50 10, 17	0000100080000000 0000000080000000 0400000000000000 0400000000000000 0000004000000000 0000004000000000 0001000080800040 0000000080000040	2^{-8}
6	25, 29 39, 43	0000000000000000 0000000000000000 0000000000000000 0000000000000000 0001000000800000 0001000000000000 0000100000000000 0000100000000000	2^{-3}
7	8, 35 56,22	0000000000000000 0000000000000000 0000000000800000 0000000000800000 0000000000000000 0000000000000000 0000000000000000 0000000000000000	2^{-1}
8		0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0400000000000000	2^{-1}
K_2		0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0000000000000000 0400000000000000	–
		no differences in rounds 9-16	
K_4		0000000000000000 0000000000000000 0000000000000000 0000000000000000 0400000000000000 0000000000000000 0000000000000000 0000000000000000	–
16	46, 36 19, 37	0000000000000000 0000000000000000 0000000000000000 0000000000000000 – 0000000000000000 0000000000000000 0000000000000000	1
17	33, 27 14, 42	0000000000000000 0000000000000000 0400000000000000 0000000000000000 0000000000000000 0400000000000000 0000000000000000 0000000000000000	2^{-2}
18	17, 49 36, 39	0400000000000000 0000000000000000 0400000000000000 0000000000000000 0000000000000000 0400000000000100 0000000000000000 0400000000000000	2^{-5}
19	44, 9 54, 56	0400000000000000 0400000000000000 0400000000000100 0400000200000000 0400000000000000 0400100040000100 0400000000000000 0400000000000000	2^{-9}
20		0000000200000100 0000004000000000 0000100040000100 0004000000000000 0000000000000000 4001100440100100 0000000000000000 0000040200000108	–

The top path we used consists of 18 rounds. Because $\varDelta K_2 = (0, 0, 0, 0, 0, 0, 0, \pm 2^{58})$ and $\varDelta K_3 = (0, 0, 0, 0, 0, 0, 0, 0)$, we select the intermediate difference $\varDelta V_8$ to meet $(0, 0, 0, 0, 0, 0, 0, \mp 2^{58})$. In this way, we get an 8-round path with zero-difference from rounds 9 to 16. By extending the difference $\varDelta V_8$ in the backward direction for 6 rounds and the difference $\varDelta \hat{V}_{16} = \varDelta K_4$ in the forward direction for 4 rounds, an 18-round differential path with high probability is obtained.

Similarly, we choose $\varDelta V_{24}$ as $(0, 0, 0, 0, 0, 0, 0, 2^{63})$ to compensate the difference $\varDelta K_6 = (0, 0, 0, 0,\ 0, 0, 0, 2^{63})$, which results in zero difference in rounds 25 to 32. As a consequence, a 18-round differential path with high probability also can be acquired by linearly expanding the difference $\varDelta V_{24}$ backward for 4 rounds and the difference $\varDelta \hat{V}_{32} = \varDelta K_8$ forward for 6 rounds.

Table 5. The bottom differential path used for boomerang attacks of Skein-512

Rd	shifts	Difference				Pr
20		0000000010004800 8000000020000200	0020001000004000 8000000020000200	0002201000080000 0000088000080000	0000200000080000 8000008000080000	2^{-7}
K_5		0000000000000000 0000000000000000	0000000000000000 8000000000000000	0000000000000000 0000000000000000	0000000000000000 8000000000000000	–
20	39, 30 34, 24	– –	0020001000004000 0000000020000200	– –	0000200000080000 0000008000080000	2^{-9}
21	13, 50 10, 17	0002001000000000 0000080000000000	0000001000000000 0000080000000000	8000000000000000 0020001010000800	8000000000000000 0000001000000800	2^{-7}
22	25, 29 39, 43	0000000000000000 0020000010000000	0000000000000000 0020000000000000	0000000000000000 0002000000000000	0000000000000000 0002000000000000	2^{-7}
23	8, 35 56, 22	0000000000000000 0000000000000000	0000000000000000 0000000000000000	0000000010000000 0000000000000000	0000000010000000 0000000000000000	2^{-3}
24		0000000000000000 0000000000000000	0000000000000000 0000000000000000	0000000000000000 0000000000000000	0000000000000000 8000000000000000	2^{-1}
K_6		0000000000000000 0000000000000000	0000000000000000 0000000000000000	0000000000000000 0000000000000000	0000000000000000 8000000000000000	
		no differences in Rounds 25-32				
K_8		0000000000000000 8000000000000000	0000000000000000 0000000000000000	0000000000000000 0000000000000000	0000000000000000 0000000000000000	–
32	46, 36 19, 37	0000000000000000 8000000000000000	0000000000000000 0000000000000000	0000000000000000 0000000000000000	0000000000000000 0000000000000000	1
33	33, 27 14, 42	0000000000000000 0000000000000000	0000000000000000 8000000000000000	8000000000000000 0000000000000000	0000000000000000 0000000000000000	1
34	17, 49 36, 39	8000000000000000 0000000000000000	0000000000000000 8000000000002000	8000000000000000 0000000000000000	0000000000000000 8000000000000000	1
35	44, 9 54, 56	8000000000000000 8000000000000000	8000000000000000 8002000800002000	8000000000002000 8000000000000000	8000004000000000 8000000000000000	2^{-1}
36		0000004000002000 0000000000000000	0000080000000000 0022008802002008	0002000800002000 0000000000000000	0080000000000000 0000804000002100	2^{-5}
K_9		0000000000000000 8000000000000000	0000000000000000 0000000000000000	0000000000000000 8000000000000000	8000000000000000 0000000000000000	
36		0000004000002000 8000000000000000	0000080000000000 0022008802002008	0002000800002000 8000000000000000	8080000000000000 0000804000002100	2^{-18}
36	39, 30 34, 24	– –	0000080000000000 0022008802002008	– –	8080000000000000 0000804000002100	2^{-13}
37	13, 50 10, 17	8082000800002000 8000804000002100	0000084000042000 882280a802882228	8022008802002008 0000084000002000	c000806100002180 8082000820202000	2^{-18}
38		402280e902000188 8082084820200000	818a084884040000 8220a0e22200a108	082200e802880328 8082084800040000	8092480860210104 062180eb03840188	2^{-45}

The two differential paths are shown in Tables 4 and 5, where we use two kinds of differences: the XOR difference and the integer modular substraction difference. In the rounds after adding the subkey, we express the differences in the positions $\hat{a}_i$, $\hat{c}_i$, $\hat{e}_i$ and $\hat{g}_i$ with the integer modular substraction difference (except the final adding key round), because the XOR operations are not included when computing the next chaining value V_{i+1}; in the other positions of the differential path, we use the XOR difference.

4.2 Message Modifications for the Middle Rounds

The conditions of the middle 8 rounds can be satisfied by the message modifications. The two pair short differentials in the boomerang distinguisher from rounds 16 to 24 are shown in Fig. 2. Let D_1, D_2 denote the top two paths from rounds 20 down to 16, and D_3, D_4 be the bottom two paths from rounds 20 to 24. Then the sufficient conditions for the four paths are shown in Table 6.

If we select the chaining variables $V_{20}^{(1)}$ and the subkey $K_5^{(1)}$ randomly, then the conditions in D_1 can be fulfilled by modifying $V_{20}^{(1)}$, and those in D_3 can be satisfied by modifying $K_5^{(1)}$. But for conditions in D_2 and D_4, we cannot correct them directly because the pairs $(V_{20}^{(3)}, K_5^{(3)})$ and $(V_{20}^{(2)}, K_5^{(2)})$ are related to the pair $(V_{20}^{(1)}, K_5^{(1)})$.

Let us focus on the 39 common non-zero difference bits for D_1 and D_2 which are generated by $(V_i^{(1)}, V_i^{(2)})$ and $(V_i^{(3)}, V_i^{(4)})$ respectively $(i = 20, 19, 18, 17)$. We force the values of $V_i^{(3)}$ in these bits to be equal to the values of $V_i^{(1)}$ in the corresponding bits by the message modifications. That is, $a_{20,9}^{(3)} = a_{20,9}^{(1)}$, $a_{20,34}^{(3)} = a_{20,34}^{(1)}$, $b_{20,39}^{(3)} = b_{20,39}^{(1)}$, $\cdots$, $c_{17,59}^{(3)} = c_{17,59}^{(1)}$, $f_{17,59}^{(3)} = f_{17,59}^{(1)}$ (see Table 7). As a result, if all the sufficient conditions for the path D_1 are satisfied, then all the conditions in D_2 must be satisfied. For the fixed input difference γ of D_3, we can easily deduce that the conditions in Table 7 can be satisfied with probability $2^{-7.4}$, which is much higher than the average probability 2^{-17}. This is also verified by our experiments. All the conditions in Table 7 can be satisfied by modifying $V_{20}^{(1)}$.

Similarly, we can convert the conditions for D_4 in Table 6 to those in Table 8. These conditions hold with probability $2^{-8.4}$ when D_1 hold, which is much better than the average probability 2^{-33}. All the conditions in Table 8 can be fulfilled by modifying $K_5^{(1)}$.

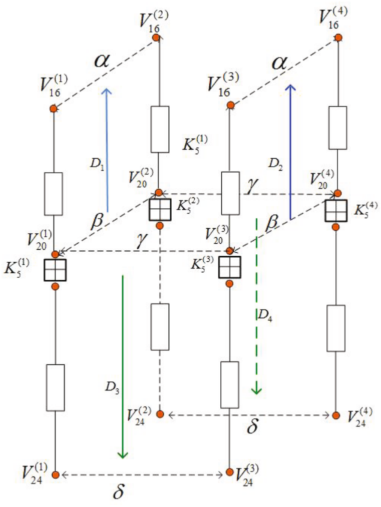


Fig. 2. The middle rounds in a boomerang distinguisher

Table 6. The conditions for differential paths D_1, D_2, D_3 and D_4

rounds	Conditions for D_1	Conditions for D_2
20	$a^{(1)}_{20,43} \oplus h^{(1)}_{20,43} = a^{(1)}_{20,34}$, $c^{(1)}_{20,63} \oplus f^{(1)}_{20,63} = c^{(1)}_{20,9}$, $c^{(1)}_{20,21} \oplus f^{(1)}_{20,21} = c^{(1)}_{20,31}$, $c^{(1)}_{20,35} \oplus f^{(1)}_{20,35} = c^{(1)}_{20,45}$	$a^{(3)}_{20,43} \oplus h^{(3)}_{20,43} = a^{(3)}_{20,34}$, $c^{(3)}_{20,63} \oplus f^{(3)}_{20,63} = c^{(3)}_{20,9}$, $c^{(3)}_{20,21} \oplus f^{(3)}_{20,21} = c^{(3)}_{20,31}$, $c^{(3)}_{20,35} \oplus f^{(3)}_{20,35} = c^{(3)}_{20,45}$
19	$a^{(1)}_{19,59} = b^{(1)}_{19,59} \oplus 1$, $c^{(1)}_{19,9} = a^{(1)}_{20,9}$, $c^{(1)}_{19,59} = d^{(1)}_{19,59} \oplus 1$, $e^{(1)}_{19,59} = f^{(1)}_{19,59} \oplus 1$, $g^{(1)}_{19,59} = h^{(1)}_{19,59} \oplus 1$, $f^{(1)}_{18,9} = c^{(1)}_{19,9}$, $f^{(1)}_{18,59} = c^{(1)}_{19,59}$, $h^{(1)}_{18,59} = e^{(1)}_{19,59}$	$a^{(3)}_{19,59} = b^{(3)}_{19,59} \oplus 1$, $c^{(3)}_{19,9} = a^{(3)}_{20,9}$, $c^{(3)}_{19,59} = d^{(3)}_{19,59} \oplus 1$, $e^{(3)}_{19,59} = f^{(3)}_{19,59} \oplus$, $g^{(3)}_{19,59} = h^{(3)}_{19,59} \oplus 1$, $f^{(3)}_{18,9} = c^{(3)}_{19,9}$, $f^{(3)}_{18,59} = c^{(3)}_{19,59}$, $h^{(3)}_{18,59} = e^{(3)}_{19,59}$
18	$a^{(1)}_{18,59} = g^{(1)}_{19,59}$, $c^{(1)}_{18,59} = a^{(1)}_{19,59}$, $f^{(1)}_{17,59} = c^{(1)}_{18,59}$	$a^{(3)}_{18,59} = g^{(3)}_{19,59}$, $c^{(3)}_{18,59} = a^{(3)}_{19,59}$, $f^{(3)}_{17,59} = c^{(3)}_{18,59}$
17	$c^{(1)}_{17,59} = a^{(1)}_{18,59}$, $k_{8,59} = c^{(1)}_{17,59}$	$c^{(3)}_{17,59} = a^{(3)}_{18,59}$, $k_{8,59} = c^{(3)}_{17,59}$
	Conditions for D_3	Conditions for D_4
20	$b^{(1)}_{20,15} = a^{(1)}_{20,15} \oplus 1$, $d^{(1)}_{20,20} = c^{(1)}_{20,20} \oplus 1$, $d^{(1)}_{20,46} = c^{(1)}_{20,46} \oplus 1$, $f^{(1)}_{20,10} = e^{(1)}_{20,10} \oplus 1$, $f^{(1)}_{20,30} = e^{(1)}_{20,30} \oplus 1$, $h^{(1)}_{20,20} = g^{(1)}_{20,20} \oplus 1$, $h^{(1)}_{20,40} = g^{(1)}_{20,40} \oplus 1$	$b^{(2)}_{20,15} = a^{(2)}_{20,15} \oplus 1$, $d^{(2)}_{20,20} = c^{(2)}_{20,20} \oplus 1$, $d^{(2)}_{20,46} = c^{(2)}_{20,46} \oplus 1$, $f^{(2)}_{20,10} = e^{(2)}_{20,10} \oplus 1$, $f^{(2)}_{20,30} = e^{(2)}_{20,30} \oplus 1$, $h^{(2)}_{20,20} = g^{(2)}_{20,20} \oplus 1$, $h^{(2)}_{20,40} = g^{(2)}_{20,40} \oplus 1$
20	$\hat{b}^{(1)}_{20,15} = b^{(1)}_{20,15}$, $\hat{b}^{(1)}_{20,37} = b^{(1)}_{20,37}$, $\hat{b}^{(1)}_{20,54} = b^{(1)}_{20,54}$, $\hat{d}^{(1)}_{20,20} = d^{(1)}_{20,20}$, $\hat{d}^{(1)}_{20,46} = d^{(1)}_{20,46}$, $\hat{f}^{(1)}_{20,10} = f^{(1)}_{20,10}$, $\hat{f}^{(1)}_{20,30} = f^{(1)}_{20,30}$, $\hat{h}^{(1)}_{20,40} = h^{(1)}_{20,40}$	$\hat{b}^{(2)}_{20,15} = b^{(2)}_{20,15}$, $\hat{b}^{(2)}_{20,37} = b^{(2)}_{20,37}$, $\hat{b}^{(2)}_{20,54} = b^{(2)}_{20,54}$, $\hat{d}^{(2)}_{20,20} = d^{(2)}_{20,20}$, $\hat{d}^{(2)}_{20,46} = d^{(2)}_{20,46}$, $\hat{f}^{(2)}_{20,10} = f^{(2)}_{20,10}$, $\hat{f}^{(2)}_{20,30} = f^{(2)}_{20,30}$, $\hat{h}^{(2)}_{20,40} = h^{(2)}_{20,40}$
21	$a^{(1)}_{21,37} = c^{(1)}_{20,37}$, $a^{(1)}_{21,50} = c^{(1)}_{20,50}$, $e^{(1)}_{21,44} = g^{(1)}_{20,44}$, $g^{(1)}_{21,12} = a^{(1)}_{20,12}$, $g^{(1)}_{21,29} = a^{(1)}_{20,29}$, $g^{(1)}_{21,37} = b^{(1)}_{20,37}$, $g^{(1)}_{21,54} = b^{(1)}_{20,54}$, $b^{(1)}_{21,37} = a^{(1)}_{21,37} \oplus 1$, $f^{(1)}_{21,44} = e^{(1)}_{21,44} \oplus 1$, $h^{(1)}_{21,12} = g^{(1)}_{21,12} \oplus 1$, $h^{(1)}_{21,37} = g^{(1)}_{21,37} \oplus 1$	$a^{(2)}_{21,37} = c^{(2)}_{20,37}$, $a^{(2)}_{21,50} = c^{(2)}_{20,50}$, $e^{(2)}_{21,44} = g^{(2)}_{20,44}$, $g^{(2)}_{21,12} = a^{(2)}_{20,12}$, $g^{(2)}_{21,29} = a^{(2)}_{20,29}$, $g^{(2)}_{21,37} = b^{(2)}_{20,37}$, $g^{(2)}_{21,54} = b^{(2)}_{20,54}$, $b^{(2)}_{21,37} = a^{(2)}_{21,37} \oplus 1$, $f^{(2)}_{21,44} = e^{(2)}_{21,44} \oplus 1$, $h^{(2)}_{21,12} = g^{(2)}_{21,12} \oplus 1$, $h^{(2)}_{21,37} = g^{(2)}_{21,37} \oplus 1$
22	$e^{(1)}_{22,29} = g^{(1)}_{21,29}$, $e^{(1)}_{22,54} = g^{(1)}_{21,54}$, $f^{(1)}_{22,54} = e^{(1)}_{22,54} \oplus 1$, $g^{(1)}_{22,50} = a^{(1)}_{21,50}$, $h^{(1)}_{22,50} = g^{(1)}_{22,50} \oplus 1$	$e^{(2)}_{22,29} = g^{(2)}_{21,29}$, $e^{(2)}_{22,54} = g^{(2)}_{21,54}$, $f^{(2)}_{22,54} = e^{(2)}_{22,54} \oplus 1$, $g^{(2)}_{22,50} = a^{(2)}_{21,50}$, $h^{(2)}_{22,50} = g^{(2)}_{22,50} \oplus 1$
23	$c^{(1)}_{23,29} = e^{(1)}_{22,29}$, $d^{(1)}_{23,29} = c^{(1)}_{23,29} \oplus 1$	$c^{(2)}_{23,29} = e^{(2)}_{22,29}$, $d^{(2)}_{23,29} = c^{(2)}_{23,29} \oplus 1$

After the message modifications, the boomerang distinguisher in the middle 8 rounds hold with probability close to 1. We also observe that the differential path D_2 is heavily dependent on D_3, and the path D_4 is heavily dependent on D_1. The reason of contradictions in the previous attacks on Skein-512 is that there exist contradict conditions in D_2 or D_4 when the paths D_1 and D_3 hold.

Table 7. The conditions for Differential Path D_2 which hold with probability $2^{-7.4}$

round	conditions	pr
20	$a^{(3)}_{20,9} = a^{(1)}_{20,9}$, $a^{(3)}_{20,34} = a^{(1)}_{20,34}$, $b^{(3)}_{20,39} = b^{(1)}_{20,39}$, $c^{(3)}_{20,9} = c^{(1)}_{20,9}$, $c^{(3)}_{20,31} = c^{(1)}_{20,31}$, $c^{(3)}_{20,45} = c^{(1)}_{20,45}$, $d^{(3)}_{20,51} = d^{(1)}_{20,51}$, $f^{(3)}_{20,9} = f^{(1)}_{20,9}$, $f^{(3)}_{20,21} = f^{(1)}_{20,21}$, $f^{(3)}_{20,31} = f^{(1)}_{20,31}$, $f^{(3)}_{20,35} = f^{(1)}_{20,35}$, $f^{(3)}_{20,45} = f^{(1)}_{20,45}$, $f^{(3)}_{20,49} = f^{(1)}_{20,49}$, $f^{(3)}_{20,63} = f^{(1)}_{20,63}$, $g^{(3)}_{20,59} = g^{(1)}_{20,9}$, $h^{(3)}_{20,4} = h^{(1)}_{20,4}$, $h^{(3)}_{20,9} = h^{(1)}_{20,9}$, $h^{(3)}_{20,34} = h^{(1)}_{20,34}$, $f^{(3)}_{20,43} = f^{(1)}_{20,43}$	1
19	$b^{(3)}_{19,59} = b^{(1)}_{19,59}$, $a^{(3)}_{19,59} = a^{(1)}_{19,59}(0.75)$, $d^{(3)}_{19,34} = d^{(1)}_{19,34}$, $d^{(3)}_{19,59} = d^{(1)}_{19,59}$, $c^{(3)}_{19,9} = c^{(1)}_{19,9}(0.87)$, $c^{(3)}_{19,59} = c^{(1)}_{19,59}(0.94)$, $f^{(3)}_{19,9} = f^{(1)}_{19,9}$, $f^{(3)}_{19,31} = f^{(1)}_{19,31}$, $f^{(3)}_{19,45} = f^{(1)}_{19,45}$, $f^{(3)}_{19,59} = f^{(1)}_{19,59}$, $e^{(3)}_{19,59} = e^{(1)}_{19,59}(0.875)$, $h^{(3)}_{19,59} = h^{(1)}_{19,59}$, $g^{(3)}_{19,59} = g^{(1)}_{19,59}(0.97)$	0.52
18	$a^{(3)}_{18,59} = a^{(1)}_{18,59}(0.687)$, $c^{(3)}_{18,59} = c^{(1)}_{18,59}(0.29)$, $f^{(3)}_{18,9} = f^{(1)}_{18,9}$, $f^{(3)}_{18,59} = f^{(1)}_{18,59}(0.25)$, $h^{(3)}_{18,59} = h^{(1)}_{18,59}(0.937)$	0.047
17	$c^{(3)}_{17,59} = c^{(1)}_{17,59}(0.5)$, $f^{(3)}_{17,59} = f^{(1)}_{17,59}(0.5)$	0.25

Table 8. The conditions for Differential Path D_4 which hold with probability $2^{-8.4}$

round	conditions	pr
20	$a^{(2)}_{20,12} = a^{(1)}_{20,12}$, $a^{(2)}_{20,15} = a^{(1)}_{20,15}$, $a^{(2)}_{20,29} = a^{(1)}_{20,29}$, $b^{(2)}_{20,15} = b^{(1)}_{20,15}$, $b^{(2)}_{20,37} = b^{(1)}_{20,37}$, $b^{(2)}_{20,54} = b^{(1)}_{20,54}$, $c^{(2)}_{20,20} = c^{(1)}_{20,20}$, $c^{(2)}_{20,37} = c^{(1)}_{20,37}$, $c^{(2)}_{20,46} = c^{(1)}_{20,46}$, $c^{(2)}_{20,50} = c^{(1)}_{20,50}$, $d^{(2)}_{20,20} = d^{(1)}_{20,20}$, $d^{(2)}_{20,46} = d^{(1)}_{20,46}$, $e^{(2)}_{20,10} = e^{(1)}_{20,10}$, $e^{(2)}_{20,30} = e^{(1)}_{20,30}$, $f^{(2)}_{20,10} = f^{(1)}_{20,10}$, $f^{(2)}_{20,30} = f^{(1)}_{20,30}$, $g^{(2)}_{20,20} = g^{(1)}_{20,20}$, $g^{(2)}_{20,40} = g^{(1)}_{20,40}$, $g^{(2)}_{20,44} = g^{(1)}_{20,44}$, $h^{(2)}_{20,20} = h^{(1)}_{20,20}$, $h^{(2)}_{20,40} = h^{(1)}_{20,40}$	1
	$\hat{b}^{(2)}_{20,15} = \hat{b}^{(1)}_{20,15}$, $\hat{b}^{(2)}_{20,37} = \hat{b}^{(1)}_{20,37}$, $\hat{b}^{(2)}_{20,54} = \hat{b}^{(1)}_{20,54}$, $\hat{d}^{(2)}_{20,20} = \hat{d}^{(1)}_{20,20}$, $\hat{d}^{(2)}_{20,46} = \hat{d}^{(1)}_{20,46}$, $\hat{f}^{(2)}_{20,10} = \hat{f}^{(1)}_{20,10}(0.5)$, $\hat{f}^{(2)}_{20,30} = \hat{f}^{(1)}_{20,30}$, $\hat{h}^{(2)}_{20,40} = \hat{h}^{(1)}_{20,40}$	0.5
21	$a^{(2)}_{21,37} = a^{(1)}_{21,37}$, $a^{(2)}_{21,50} = a^{(1)}_{21,50}(0.97)$, $b^{(2)}_{21,37} = b^{(1)}_{21,37}$, $e^{(2)}_{21,44} = e^{(1)}_{21,44}(0.5)$, $f^{(2)}_{21,44} = f^{(1)}_{22,44}$, $g^{(2)}_{21,12} = g^{(1)}_{21,12}(0.875)$, $g^{(2)}_{21,29} = g^{(1)}_{21,29}$, $g^{(2)}_{21,37} = g^{(1)}_{21,37}(0.875)$, $g^{(2)}_{21,54} = g^{(1)}_{21,54}$, $h^{(2)}_{21,12} = h^{(1)}_{21,12}(0.875)$, $h^{(2)}_{21,37} = h^{(1)}_{21,37}$	0.32
22	$e^{(2)}_{22,29} = e^{(1)}_{22,29}(0.84)$, $e^{(2)}_{22,54} = e^{(1)}_{22,54}(0.75)$, $f^{(2)}_{22,54} = f^{(1)}_{22,54}(0.5)$, $g^{(2)}_{22,50} = g^{(1)}_{22,50}(0.97)$, $h^{(2)}_{22,50} = h^{(1)}_{22,50}(0.5)$	0.15
23	$c^{(2)}_{23,29} = c^{(1)}_{23,29}(0.24)$, $d^{(2)}_{23,29} = d^{(1)}_{23,29}(0.5)$	0.12

4.3 Complexity of the Attack

Using the differential paths given in Table 4 and Table 5, we can construct a boomerang distinguisher for Skein-512 reduced to 32 rounds (out of 72 rounds). The top path in the backward direction (rounds 16-4) holds with probability 2^{-37} after the message modifications. The bottom path in the forward direction (rounds 20-36) holds with probability 2^{-24} after message modifications.

So the complexity of the 32-round boomerang distinguisher is $2^{2\cdot(37+24)} = 2^{122}$ by using the differential paths in Table 4 and 5. It can be reduced to $2^{2\cdot(13+6)} \times 3^{24+18} \approx 2^{104.5}$ if we only want $\bigoplus_{i=1}^{4} P_i = 0$ and $\bigoplus_{i=1}^{4} C_i = 0$,

Table 9. A quartet that satisfies the paths for rounds 5-36 without the initial and final subkeys

Message of Round 5				
$M^{(1)}$	efeffeca89966f57	b9ede50911910872	b80346f52e40f9b2	413a42e591e3d564
	b854665ac709fdc1	5b81218db8689f63	1454025d1e252a79	40086ca8b43d3382
$M^{(2)}$	efefeecb09966f57	b9ede50891910872	b40346f52e40f9b2	453a42e591e3d564
	b854661ac709fdc1	5b8121cdb8689f63	1455025d9ea52a39	40086ca8343d33c2
$M^{(3)}$	5b44c68c6c74d8d8	462dcb0d8f65c514	4660e299d27ed556	1622a67e6860f1b3
	8631f78ea11186d9	29bf5dee4c4708bf	54cb280ae171a9fd	df5814e7668fdf95
$M^{(4)}$	5b44d68dec74d8d8	462dcb0c0f65c514	4a60e299d27ed556	1222a67e6860f1b3
	8631f7cea11186d9	29bf5dae4c4708bf	54ca280a61f1a9bd	df5814e7e68fdfd5
Key				
$K^{(1)}$	fd4707e3dc7b1c35	3f64c6f0bd13466a	45e7c90173366b70	dc71a6f93dbfc9d5
	5c977a7bbc2dbe6d	56889bd71af7189f	8bc7bcb9d86167a1	0091f15b4d1aeaee
$K^{(2)}$	f94707e3dc7b1c35	3f64c6f0bd13466a	45e7c90173366b70	dc71a6f93dbfc9d5
	5c977a7bbc2dbe6d	56889bd71af7189f	8bc7bcb9d86167a1	0091f15b4d1aeaee
$K^{(3)}$	fd4707e3dc7b1c35	3f64c6f0bd13466a	45e7c90173366b70	5c71a6f93dbfc9d5
	dc977a7bbc2dbe6d	56889bd71af7189f	8bc7bcb9d86167a1	0091f15b4d1aeaee
$K^{(4)}$	f94707e3dc7b1c35	3f64c6f0bd13466a	45e7c90173366b70	5c71a6f93dbfc9d5
	dc977a7bbc2dbe6d	56889bd71af7189f	8bc7bcb9d86167a1	0091f15b4d1aeaee
Tweak				
$T^{(1)}, T^{(2)}$	55422f07b9ea59be	511ad7aa13272cc9	51422f07b9ea59be	551ad7aa13272cc9
$T^{(3)}, T^{(4)}$	55422f07b9ea59be	d11ad7aa13272cc9	51422f07b9ea59be	d51ad7aa13272cc9

Table 10. A quartet that satisfies the paths for rounds 8-36 including the initial and final subkeys

Message of Round 8				
$M^{(1)}$	81eb65560efb565c	42171413b9dae252	ba7f35e83ceec8b7	d5dbcf318a0ecf74
	5d1c176606c51b45	4f8fc8fc188100d4	45d34efc985185f5	673059aaf448427c
$M^{(2)}$	81eb65560efb565c	42171413b9dae252	ba7f35e83ceec8b7	d5dbcf318a0ecf74
	5d1c176606c51b45	4f8fc8fc188100d4	45d34efc985185f5	6b3059aaf448427c
$M^{(3)}$	f96c2ea16f7aa900	7dbe4b7cc9bef8ea	f94e7e6cff763332	f44decb0fcb6ecac
	7f30973fad83191f	94591dff30d2e161	74c7323813fc5c42	54e6ccf74a6a1d11
$M^{(4)}$	f96c2ea16f7aa900	7dbe4b7cc9bef8ea	f94e7e6cff763332	f44decb0fcb6ecac
	7f30973fad83191f	94591dff30d2e161	74c7323813fc5c42	58e6ccf74a6a1d11
Key				
$K^{(1)}$	bf07320940fa73f1	64561111c05cc195	bbf500154032fa6d	8dff001fb0239bbf
	5e36a0172124dd89	50e99cdbc81bab42	3ac1c8825115600a	12b40efea4188dab
$K^{(2)}$	bb07320940fa73f1	64561111c05cc195	bbf500154032fa6d	8dff001fb0239bbf
	5e36a0172124dd89	50e99cdbc81bab42	3ac1c8825115600a	12b40efea4188dab
$K^{(3)}$	bf07320940fa73f1	64561111c05cc195	bbf500154032fa6d	0dff001fb0239bbf
	de36a0172124dd89	50e99cdbc81bab42	3ac1c8825115600a	12b40efea4188dab
$K^{(4)}$	bb07320940fa73f1	64561111c05cc195	bbf500154032fa6d	0dff001fb0239bbf
	de36a0172124dd89	50e99cdbc81bab42	3ac1c8825115600a	12b40efea4188dab
Tweak				
$T^{(1)}, T^{(2)}$	8fe4eab7841221ae	82aeedc8d61e677b	8be4eab7841221ae	86aeedc8d61e677b
$T^{(3)}, T^{(4)}$	8fe4eab7841221ae	02aeedc8d61e677b	8be4eab7841221ae	06aeedc8d61e677b

Table 11. The modified differential path in the middle rounds used for boomerang attacks of Skein-512 in [1]

Rd	shifts	The Difference for the top path from rounds 12-16			
K_3		0000000000000000	0000000000000000	0000000000000000	0000000000000000
		8000000000000000	0000000000000000	0000000000000000	0000000000000000
12	33, 49	0000000000000000	0000000000000000	0000000000000000	0000000000000000
	8, 42	8000000000000000	0000000000000000	0000000000000000	0000000000000000
13	39, 27	0000000000000000	0000000000000000	8000000000000000	0000000000000000
	41, 14	0000000000000000	8000000000000000	0000000000000000	0000000000000000
14	29, 26	8000000000000000	0000000000000000	8000000000000000	0000000000000000
	11, 9	0000000000000000	8000010000000000	0000000000000000	8000000000000000
15	33, 51	8000000000000000	8000000000000000	8000010000000000	8000000000000100
	39, 35	8000000000000000	8008010000000400	8000000000000000	8000000000000000
16		0000010000000100	0000000100000000	0008010000000400	0000000400000000
		0000000000000000	000a014004008400	0000000000000000	0004010000000100
K_4		0000000000000000	0000000000000000	0000000000000000	8000000000000000
		8000000000000000	0000000000000000	0000000000000000	8000000000000000
16		0000010000000100	0000000100000000	0008010000000400	8000000400000000
		8000000000000000	000a014004008400	0000000000000000	0804010000000100
		The Difference for the bottom path from rounds 16-20			
16	38, 30	4008401080102024	4000400080002004	0440018001000400	0440008000000400
	50, 53	0000000000040090	0000000000040080	0200000000008010	0000000000008010
17	48, 20	0000010001000000	0000010000000000	0000000000000010	0000000000000010
	43, 31	0200000000000000	0200000000000000	0008001000100020	0000000000100020
18	34, 14	0000000000000000	0000000000000000	0000000000000000	0000000000000000
	15, 27	0008001000000000	0000001000000000	0000000001000000	0000000001000000
19	26, 12	0000000000000000	0000000000000000	0008000000000000	0008000000000000
	58, 7	0000000000000000	0000000000000000	0000000000000000	0000000000000000
20		0000000000000000	0000000000000000	0000000000000000	0000000000000000
		0000000000000000	0000000000000000	0000000000000000	8000000000000000
K_5		0000000000000000	0000000000000000	0000000000000000	0000000000000000
		0000000000000000	0000000000000000	0000000000000000	8000000000000000

because the probability for $\bigoplus_{i=1}^{4} x_j^{(i)} = 0$ is about $1/3$ where $x_j^{(i)}$ denote the non-zero difference bits in rounds 4 and 36.

Extending the 32-round boomerang distinguisher for two more rounds in the backward and forward directions, we can get the 33-/34-/35-/36-round boomerang distinguisher on Skein-512 as follows:

– The complexity of 33-round distinguisher (rounds 4-37) is about $2^{2\cdot(13+6)} \times 3^{24+13+18} \approx 2^{125}$.
– The complexity of 34-round distinguisher (rounds 3-37) is about $2^{2\cdot(37+6)} \times 3^{35+13+18} \approx 2^{190.6}$.
– The complexity of 35-round distinguisher (rounds 3-38) is about $2^{2\cdot(72+82)} = 2^{308}$.
– The complexity of 36-round distinguisher (rounds 2-38) is about $2^{2\cdot(72+82+73)} = 2^{454}$.

Remark: For the 32-/33-/34-round attacks, we use the Type III boomerang distinguisher, the complexity for the best algorithm is 2^{256}; for the 35-/36-round attacks, we use the Type I boomerang distinguisher, the generic complexity is about 2^{512}. Note that the initial and final key-additions are included in our 32-round reduced Skein-512 but they are not included in the distinguishers for 33 to 36 rounds.

In the following, we give examples of the quartets to show that our technique used for 32 to 36 rounds attack is valid. Table 9 gives a zero-sum quartet for rounds 5-36 of Skein-512 (the initial and final subkeys are not included) with $\bigoplus_{i=1}^{4} V_5^{(i)} = 0$ and $\bigoplus_{i=1}^{4} V_{36}^{(i)} = 0$. The complexity of the attack is about 2^{32}. Table 10 gives a zero-sum quartet for rounds 8-36 of Skein-512 with $\bigoplus_{i=1}^{4} V_8^{(i)} = 0$ and $\bigoplus_{i=1}^{4} \hat{V}_{36}^{(i)} = 0$ (the initial and final subkeys are included). The complexity of the attack is about $2^{40.5}$.

4.4 The Incompatibility of Previous Boomerang Attacks on Threefish-512

In papers [1,2], Aumasson *et al.* first presented the boomerang distinguishers on Threefish-512 reduced to 35 rounds. We studied the differential paths used to boomerang attack in Tables 6 and 7 of [2], and found that they used an inverse permutation instead of the original one. We correct the permutation and give the middle 8-round differential paths (see Table 11) using the differences for the key words and tweaks proposed in [1] under the old rotation constants. For the top path, the MSB differences are set in k_7 and t_0. And for the bottom path, the MSB differences are set in k_2, k_3, t_0 and t_1.

From the bottom path, it is easy to deduce that $\hat{d}_{16,11}^{(1)} = \hat{c}_{16,11}^{(1)} \oplus 1$, $\hat{d}_{16,11}^{(2)} = \hat{c}_{16,11}^{(2)} \oplus 1$. From the top path, we know that $\hat{d}_{16,11}^{(1)} = \hat{d}_{16,11}^{(2)}$, so we get $\hat{c}_{16,11}^{(1)} = \hat{c}_{16,11}^{(2)}$. But from the top path, it's obvious that $\hat{c}_{16,11}^{(1)} = \hat{c}_{16,11}^{(2)} \oplus 1$. Hence a contradiction appears. Similarly, the differences on bit 41 for the top and bottom paths are also incompatible.

5 Key Recovery Attack on Reduced Threefish-512

Our boomerang distinguishers for 32 to 34 rounds Skein-512 are also applicable to (related) key recovery attack on Threefish-512. In this case, the complexity for the middle 8 rounds are added, and the initial and final subkeys are included. For the fixed input and output differences α and γ, the probabilities of the boomerang distinguishers for Threefish-512 reduced to 32(rounds 4-36), 33(rounds 4-37), 34(rounds 3-37) rounds are 2^{-177}, 2^{-301} and 2^{-419} respectively.

Consequently, we can mount key recovery attacks on reduced Threefish-512 for 1/4 of the key space, with time complexities 2^{181}, 2^{305} and 2^{424}, respectively. We give the procedure of the key recovery attack on 32-round Threefish-512 as an example.

1. For $i = 1, ..., 2^{179}$
   (a) Randomly choose plaintext P_1^i, compute $P_2^i = P_1^i \oplus \alpha$.
   (b) Encrypt plaintext pair (P_1^i, P_2^i) with $K^{(1)}, K^{(2)}$ respectively to get (C_1^i, C_2^i). Compute $C_3^i = C_1^i \oplus \delta, C_4^i = C_2^i \oplus \delta$. Then decrypt (C_3^i, C_4^i) with $K^{(3)}, K^{(4)}$ respectively to get (P_3^i, P_4^i).
   (c) Check whether $P_3^i \oplus P_4^i = \alpha$, if so, store the quartet $(C_1^i, C_2^i, C_3^i, C_4^i)$.
2. (a) Guess 128 bits of the final subkey words $K_{9,a}, K_{9,b}$ and subtract them with the corresponding words of each element of quartets stored in Step 1. If for all the quartets, whose resulting words satisfy that the XOR differences before the key addition, we store this 128-bit subkey pair $(K_{9,a}, K_{9,b})$.
   (b) Similarly, sequently guess $(K_{9,c}, K_{9,d})$ and $(K_{9,f}, K_{9,h})$ and check whether the required conditions are satisfied. If yes, store the corresponding key words.
3. Search the remaining 128 bits of the final subkey by brute force.

The complexity is dominated by Step 1, which is about 2^{181} 32-round encryptions. The expected number of quartets passed Step 2(a) for a false key is $4 \times 2^{-6} = 2^{-4}$. Let Y be the number of the quartets passed Step 2(a) for a false key, using the Poisson distribution, we have $Pr(Y \geq 4) \approx 0$. The expected quartets passed Step 2(a) for the right key is 4. Let Z be the number of the quartets passed Step 2(a) for the right key, $Pr(Z \geq 4) \approx 0.9$. The success rate of Step 2(b) is similar.

6 Conclusions

In this paper, we apply the boomerang attack to distinguish the compression function of Skein-512 reduced to 36 (out of 72) rounds from a random function. We select the key difference in the 59-*th* bit instead of the difference in the most significant bit to avoid the contradiction in the previous attack for boomerang attacks on Threefish-512. We also point out that the differential paths used in the boomerang distinguisher in the middle rounds are not independent. Our boomerang distinguishers are applicable to the key recovery attack for Threefish-512 reduced to 34 rounds. Future works on Skein-512 might apply the rebound attack [8] to Threefish, although it looks very difficult to combine two short differential paths to a long one.

References

1. Aumasson, J.-P., Çalık, Ç., Meier, W., Özen, O., Phan, R.C.-W., Varıcı, K.: Improved Cryptanalysis of Skein. In: Matsui, M. (ed.) ASIACRYPT 2009. LNCS, vol. 5912, pp. 542–559. Springer, Heidelberg (2009)
2. Aumasson, J., et al.: Improved Cryptanalysis of Skein, http://eprint.iacr.org/2009/438.pdf

3. 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)
4. Biham, E., Dunkelman, O., Keller, N.: Related-Key Boomerang and Rectangle Attacks. In: Cramer, R. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 507–525. Springer, Heidelberg (2005)
5. Biryukov, A., Lamberger, M., Mendel, F., Nikolić, I.: Second-Order Differential Collisions for Reduced SHA-256. In: Lee, D.H. (ed.) ASIACRYPT 2011. LNCS, vol. 7073, pp. 270–287. Springer, Heidelberg (2011)
6. Chen, J., Jia, K.: Improved Related-Key Boomerang Attacks on Round-Reduced Threefish-512. In: Kwak, J., Deng, R.H., Won, Y., Wang, G. (eds.) ISPEC 2010. LNCS, vol. 6047, pp. 1–18. Springer, Heidelberg (2010)
7. Ferguson, N., Lucks, S., Schneier, B., Whiting, D., Bellare, M., Kohno, T., Callas, J., Walker, J.: The Skein Hash Function Family, `http://www.schneier.com/skein1.3.pdf`
8. Mendel, F., Rechberger, C., Schläffer, M., Thomsen, S.S.: The Rebound Attack: Cryptanalysis of Reduced Whirlpool and `Grøstl`. In: Dunkelman, O. (ed.) FSE 2009. LNCS, vol. 5665, pp. 260–276. Springer, Heidelberg (2009)
9. Mendel, F., Rechberger, C., Schläffer, M., Thomsen, S.S.: Rebound Attacks on the Reduced `Grøstl` Hash Function. In: Pieprzyk, J. (ed.) CT-RSA 2010. LNCS, vol. 5985, pp. 350–365. Springer, Heidelberg (2010)
10. Khovratovich, D., Nikolić, I.: Rotational Cryptanalysis of ARX. In: Hong, S., Iwata, T. (eds.) FSE 2010. LNCS, vol. 6147, pp. 333–346. Springer, Heidelberg (2010)
11. Khovratovich, D., Nikolić, I., Rechberger, C.: Rotational Rebound Attacks on Reduced Skein. In: Abe, M. (ed.) ASIACRYPT 2010. LNCS, vol. 6477, pp. 1–19. Springer, Heidelberg (2010)
12. Khovratovich, D., Rechberger, C., Savelieva, A.: Bicliques for Preimages: Attacks on Skein-512 and the SHA-2 Family. In: Canteaut, A. (ed.) FSE 2012. LNCS, vol. 7549, pp. 244–263. Springer, Heidelberg (2012)
13. 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)
14. Leurent, G., Roy, A.: Boomerang Attacks on Hash Function Using Auxiliary Differentials. In: Dunkelman, O. (ed.) CT-RSA 2012. LNCS, vol. 7178, pp. 215–230. Springer, Heidelberg (2012)
15. Leurent, G.: ARXtools: A toolkit for ARX analysis. In: The 3rd SHA-3 Conference
16. Su, B.Z., Wu, W.L., Wu, S., Dong, L.: Near-Collisions on the Reduced-Round Compression Functions of Skein and BLAKE. In: Heng, S.-H., Wright, R.N., Goi, B.-M. (eds.) CANS 2010. LNCS, vol. 6467, pp. 124–139. Springer, Heidelberg (2010)
17. Wagner, D.: The Boomerang Attack. In: Knudsen, L.R. (ed.) FSE 1999. LNCS, vol. 1636, pp. 156–170. Springer, Heidelberg (1999)
18. Wagner, D.: A Generalized Birthday Problem. In: Yung, M. (ed.) CRYPTO 2002. LNCS, vol. 2442, pp. 288–303. Springer, Heidelberg (2002)
19. 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)
20. 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)
21. Yu, H.B., Chen, J.Z., Jia, K.T., Wang, X.Y.: Near-Collision Attack on the Step-Reduced compression Function of Skein-256. Cryptology ePrint Archive, Report 2011/148 (2011), `http://eprint.iacr.org`

Boomerang and Slide-Rotational Analysis of the SM3 Hash Function

Aleksandar Kircanski[1], Yanzhao Shen[2], Gaoli Wang[2,3,⋆], and Amr M. Youssef[1]

[1] Concordia University
Concordia Institute for Information Systems Engineering
Montreal, Quebec, Canada
[2] Donghua University
School of Computer Science and Technology, Shanghai, China
[3] State Key Laboratory of Information Security
Institute of Software, Chinese Academy of Sciences, Beijing, China
wanggaoli@dhu.edu.cn

Abstract. SM3 is a hash function, designed by Xiaoyun Wang *et al.* and published by the Chinese Commercial Cryptography Administration Office for the use of electronic authentication service system. The design of SM3 builds upon the design of the SHA-2 hash function, but introduces additional strengthening features. In this paper, we present boomerang distinguishers for the SM3 compression function reduced to 32 steps out of 64 steps with complexity $2^{14.4}$, 33 steps with complexity $2^{32.4}$, 34 steps with complexity $2^{53.1}$ and 35 steps with complexity $2^{117.1}$. Examples of zero-sum quartets for the 32-step and 33-step SM3 compression function are provided. We also point out a slide-rotational property of SM3-XOR, which exists due to the fact that constants used in the steps are not independent.

Keywords: Cryptanalysis, Boomerang attack, Rotational attack, Slide attack, SM3.

1 Introduction

In December of 2007, the Chinese National Cryptographic Administration Bureau released the specification of a Trusted Cryptography Module detailing a cryptoprocessor to be used within the Trusted Computing framework in China. The module specifies a set of cryptographic algorithms that includes the SMS4 block cipher, the SM2 asymmetric algorithm and SM3, a new cryptographic hash function designed by Xiaoyun Wang *et al.* [1]. The design of SM3 resembles the design of SHA-2 but includes additional fortifying features such as feeding two message-derived words into each step, as opposed to only one in the case of SHA-2.

The only previous work that we are aware of on the analysis of SM3 has been presented by Zou *et al.* [2] at ICISC 2011 where a preimage attack on step-reduced SM3 is provided. In particular, Zou *et al.* presented attacks on SM3 reduced to 30 steps, starting from the 7-th step, with time complexity 2^{249} and 28 steps, starting from the 1-st step with time complexity $2^{241.5}$.

⋆ Corresponding author.

The use of zero-sums as distinguishers have been introduced by Aumasson *et al.* [3]. The boomerang attack [4], originally introduced for block ciphers, has been adapted to the hash function setting independently by Biryukov *et al.* [5], and Lamberger and Mendel [6]. In particular, in [5], a distinguisher for the 7-round BLAKE-32 was provided, whereas in [6] a distinguisher for the 46-step SHA-2 compression function was provided. The latter SHA-2 result was extended to 47 steps in [7]. In [8], Mendel and Nad presented boomerang distinguishers for the SIMD-512 compression function. Sasaki [9] gave a boomerang distinguisher on the full compression function of 5-pass HAVAL. Sasaki also proposed a 2-dimension sums attack on 48-step RIPEMD-128 and 42-step RIPEMD-160 in [10]. Boomerang distinguishers have also been applied to Skein and Threefish. In [11] Aumasson *et al.* proposed a related-key boomerang distinguisher on 34-step Skein and a known-related-key boomerang distinguisher on 35-step Skein. In [12], Leurent and Roy showed that, under some conditions, three independent paths instead of two can be combined to achieve a distinguisher for the compression function with complexity 2^{114}. In [13] Chen *et al.* proposed related-key boomerang distinguishers on 32-step, 33-step and 34-step Threefish-512. Recently, Yu *et al.* [14] proposed boomerang attacks on the 32-step, 33-step and 34-step Skein-512.

Khovratovich *et al.* introduced rotational distinguishers in [15], where two words are said to be rotational if they are equal up to bit-wise rotation by some number of positions. Slide attacks were introduced by Biryukov *et al.* [16] and subsequently were applied to many cryptographic primitives.

Our Contribution. In the first part of this paper, we present a boomerang attack on the SM3 hash function reduced to 32 steps out of 64 steps with complexity $2^{14.4}$, 33 steps with complexity $2^{32.4}$, 34 steps with complexity $2^{53.1}$ and 35 steps with complexity $2^{117.1}$. Particular examples of the boomerang distinguisher for the 32-step and also the 33-step compression function are provided. The previous results and a summary of ours are given in Table 1.

In the second part of the paper, we present a slide-rotational property of SM3 and we analyze the SM3-XOR compression function, which is the SM3 compression function with the addition mod 2^{32} replaced by XOR. In particular, we show that, for SM3-XOR, one can easily construct input-output pairs satisfying a simple rotational property. Such a property exists due to the fact that, unlike in SHA-2, the constants in steps i, $i+1$, for $i = 0, \ldots, 63$, $i \neq 15$ are computed by bitwise rotation starting from two predefined independent values. Previously, SHA2-XOR was analyzed in [17].

Paper Outline. The rest of the paper is organized as follows. In Section 2, we briefly review the specifications of the SM3 hash function and give the notation used in this paper. A brief overview of boomerang attacks is provided in Section 3. The differential characteristics, and a description of the boomerang attack process and its complexity evaluation are provided in Section 4. The slide-rotational property is explained in Section 5. Finally, our conclusion is given in Section 6.

Table 1. Summary of the attacks on the SM3 compression function (CF) and hash function (HF)

Attack	CF/HF	Steps	Complexity	Reference
Preimage attack	HF	28	$2^{241.5}$	[2]
Preimage attack	HF	30	2^{249}	[2]
Boomerang attack	CF	32	$2^{14.4}$	Section 4
Boomerang attack	CF	33	$2^{32.4}$	Section 4
Boomerang attack	CF	34	$2^{53.1}$	Section 4
Boomerang attack	CF	35	$2^{117.1}$	Section 4

2 Description of SM3 and Notation

In this section, we briefly review relevant specifications of the SM3 hash function and provide the notation used throughout the paper.

2.1 Description of SM3

The SM3 hash function compresses messages of arbitrary length into 256-bit hash values. Given any message, the algorithm first pads it into a message of length that is a multiple of 512 bits. We omit the padding method here since it is irrelevant to our attack. For our purpose, SM3 consists mainly of two parts: the message expansion and the state update transformation. In here, we briefly review the relevant specifications of these two components. For a detailed description of the hash function, we refer the reader to [1].

Message Expansion. The message expansion of SM3 splits the 512-bit message block M into 16 words m_i, $(0 \le i \le 15)$, and expands them into 68 expanded message words w_i $(0 \le i \le 67)$ and 64 expanded message words $w_i{}'(0 \le i \le 63)$ as follows:

$$w_i = \begin{cases} m_i, & 0 \le i \le 15, \\ P_1(w_{i-16} \oplus w_{i-9} \oplus (w_{i-3} \lll 15)) \oplus (w_{i-13} \lll 7) \oplus w_{i-6}, & 16 \le i \le 67, \end{cases}$$

$$w_i{}' = w_i \oplus w_{i+4}, 0 \le i \le 63.$$

The functions $P_0(X)$ which is used in the state update transformation and $P_1(X)$ which is used in message expansion are given by

$$P_0(X) = X \oplus (X \lll 9) \oplus (X \lll 17),$$
$$P_1(X) = X \oplus (X \lll 15) \oplus (X \lll 23).$$

State Update Transformation. The state update transformation starts from an initial value $IV = (A_0, B_0, C_0, D_0, E_0, F_0, G_0, H_0)$ of eight 32-bit words and updates them in 64 steps. In step $i + 1 (0 \le i \le 63)$ the 32-bit words w_i and $w_i{}'$ are used to update the state variables $A_i, B_i, C_i, D_i, E_i, F_i, G_i, H_i$ as follows:

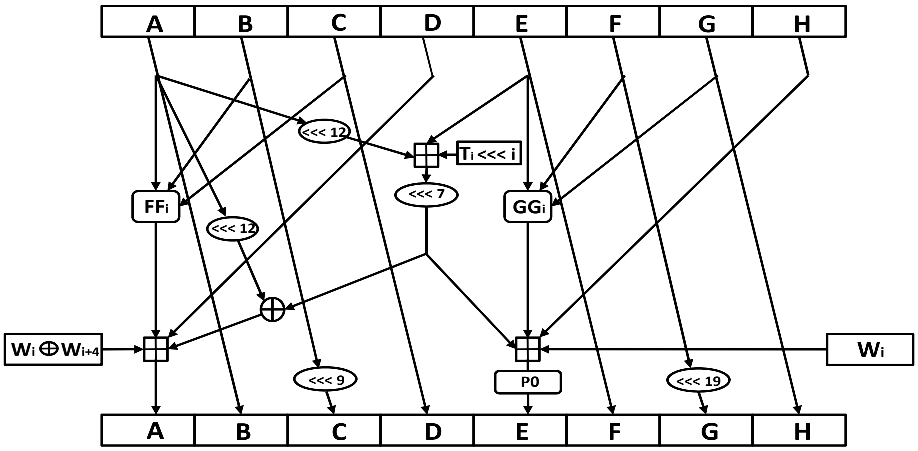


Fig. 1. One step of the SM3 hash function

$$
\begin{aligned}
SS1_i &= ((A_i \lll 12) + E_i + (T_i \lll i)) \lll 7,\\
SS2_i &= SS1_i \oplus (A_i \lll 12),\\
TT1_i &= FF_i(A_i, B_i, C_i) + D_i + SS2_i + w_i',\\
TT2_i &= GG_i(E_i, F_i, G_i) + H_i + SS1_i + w_i,\\
A_{i+1} &= TT1_i, B_{i+1} = A_i, C_{i+1} = (B_i \lll 9), D_{i+1} = C_i,\\
E_{i+1} &= P_0(TT2_i), F_{i+1} = E_i, G_{i+1} = (F_i \lll 19), H_{i+1} = G_i.
\end{aligned}
$$

The round constants are $T_i = 0x79cc4519$ for $i \in \{0, ..., 15\}$ and $T_i = 0x7a879d8a$, for $i \in \{16, ..., 63\}$. As for the bitwise Boolean functions $FF(X, Y, Z)$ and $GG(X, Y, Z)$ used in each step, we have

$$
FF(X, Y, Z) = \begin{cases} X \oplus Y \oplus Z, & 0 \le i \le 15,\\ (X \wedge Y) \vee (X \wedge Z) \vee (Y \wedge Z), & 16 \le i \le 63, \end{cases}
$$
$$
GG(X, Y, Z) = \begin{cases} X \oplus Y \oplus Z, & 0 \le i \le 15,\\ (X \wedge Y) \vee (\neg X \wedge Z), & 16 \le i \le 63. \end{cases}
$$

If M is the last block, then $(A_{64} \oplus A_0, B_{64} \oplus B_0, C_{64} \oplus C_0, D_{64} \oplus D_0, E_{64} \oplus D_0, F_{64} \oplus F_0, G_{64} \oplus G_0, H_{64} \oplus H_0)$ is the hash value. Otherwise $(A_{64} \oplus A_0, B_{64} \oplus B_0, C_{64} \oplus C_0, D_{64} \oplus D_0, E_{64} \oplus D_0, F_{64} \oplus F_0, G_{64} \oplus G_0, H_{64} \oplus H_0)$ constitutes the input of the next message block. One step of the SM3 compression function is depicted in Fig. 1.

2.2 Notation

Our attacks use the integer modular subtraction difference. In here, we introduce the notation used in throughout the rest of the paper.

1. X, Y, X' and Y' represent four 256-bit middle chaining values.
2. $IV_X, IV_Y, IV_{X'}$ and $IV_{Y'}$ represent four 256-bit initial values.
3. $H_X, H_Y, H_{X'}$ and $H_{Y'}$ represent four 256-bit outputs of the compression function.
4. $M_X, M_Y, M_{X'}$ and $M_{Y'}$ represent four 512-bit message blocks.
5. $w_{i,j}$ denotes the j-th bit of w_i, $0 \leq j \leq 31$.
6. $(W_t)_i^j$ (t can be X, Y, X' or Y') denotes $(j - i + 1)$ 32-bit words, w_i to w_j, where $i < j$.
7. $S_i[+j] = S_i + 2^j$ with no bit carry, $S_i[-j] = S_i - 2^j$ with no bit carry, where S can be $w, A, B, C, D, E, F, G, H$.
8. $S_i^{M_c}$ denotes the chaining value used in step i combined with the middle chaining value M_c where M_c can be X, Y, X', Y', and S can be $w, w', A, B, C, D, E, F, G, H, SS1$, or $SS2$.

3 Boomerang Distinguishers for Hash Functions

In this section, we review known-related-key boomerang attacks which can be used to distinguish a given permutation from a random oracle. We concentrate on the known-related-key boomerang attack to the compression function in the Davies-Mayer mode, i.e., $CF(M, K) = E(M, K) \oplus M$. As noted in [5,7,9,14], we can start from middle steps to construct boomerang distinguishers. Then we have

$$CF_0^{-1}(X, K_1) \oplus CF_0^{-1}(X \oplus \beta, K_2) = \alpha, \tag{1}$$

and

$$CF_1(X, K_1) \oplus CF_1(X \oplus \gamma, K_3) = \delta, \tag{2}$$

where the differential in CF_0^{-1} holds with probability p_0 and holds with probability p_1 in CF_1. Using these two differentials, we can construct the boomerang attack for the compression function CF as follows:

1. Choose a random value X, compute the corresponding value $X' = X \oplus \beta, Y = X \oplus \gamma, Y' = Y \oplus \beta$ and $K_2 = K_1 \oplus \beta_k, K_3 = K_1 \oplus \gamma_k, K_4 = K_3 \oplus \beta_k$.
2. Compute backward from $(X, K_1), (X', K_2), (Y, K_3), (Y', K_4)$ using CF_0^{-1} to obtain P, P', Q, Q'.
3. Compute forward from $(X, K_1), (X', K_2), (Y, K_3), (Y', K_4)$ using CF_1 to obtain C, C', D, D'.
4. Check whether $P \oplus P' = Q \oplus Q' = \alpha$ and $C \oplus D = C' \oplus D' = \delta$.

From (1) and (2),

$$P \oplus P' = Q \oplus Q' = \alpha \textit{ and } C \oplus D = C' \oplus D' = \delta, \tag{3}$$

holds with probability at least p_0^2 in the backward direction and with probability at least p_1^2 in the forward direction. Hence, assuming that the differentials are independent, the attack succeeds with probability $p_0^2 p_1^2$. The expected number of solutions to (3) is 1, if we repeat the attack about $1/(p_0^2 p_1^2)$ times.

For an n-bit random permutation, there exist 3 types of boomerang distinguishers which are summarized by Yu *et al.* in [14]. Here we recall the three distinguishers as follows.

- Type I: A quartet that satisfies $P \oplus P' = Q \oplus Q' = \alpha$, and $C \oplus D = C' \oplus D' = \delta$ where the differences α and δ are fixed. In this case, there exists a generic attack with complexity 2^n.
- Type II: A quartet that only satisfies the condition $C \oplus D = C' \oplus D'$. This type of attack is called second-order differential collision attack or zero-sum attack. In this case, we can use Wagner's generalized birthday attack [19] to obtain a quartet with the complexity $2^{n/3}$.
- Type III: A quartet satisfies the conditions $P \oplus P' = Q \oplus Q'$ and $C \oplus D = C' \oplus D'$. The complexity of this attack is about $2^{n/2}$.

In this paper, we apply a type III attack to develop distinguishers for 32/33/34/35 steps of the SM3 compression function. Therefore, the attack is valid if $p_0^2 \cdot p_1^2 > 2^{-n/2}$.

4 Attacks on the SM3 Compression Function

In this section, we describe the proposed boomerang attack on the SM3 compression function reduced to 32 steps, and then expend our attack to 33, 34 and 35 steps. Firstly, we give a summary of the differential characteristics to be used to distinguish the target compression function from random functions. Secondly, we describe how to use the message modification technique to correct the conditions in the intermediate steps by modifying the chaining values A_{16} to H_{16}. We express our attack algorithm on 32-step SM3 compression function in the third part of this section. Then we evaluate the complexity of our attack and extend it to 33, 34 and 35 steps.

4.1 Differential Characteristics

In here, we mainly describe the differential characteristics which are used to attack 32-step SM3 compression function. In Table 2, we present a differential characteristic in the backward direction from step 16 to step 1 which holds with probability 2^{-67}, and the sufficient conditions that ensure that this characteristic holds. A differential characteristic in the forward direction from step 17 to step 32 which holds with probability 2^{-34} and its associated sufficient conditions are presented in Table 3.

Finding the differential characteristics for both backward and forward directions is an important part of the attack. We construct the differential characteristics as follows.

- The characteristic has a single bit difference in the message word w_i at some step, i, followed by 15 message words without differences. When using such characteristic, 12 steps (the ones that follow i) can be bypassed with probability 1. Because of the fast diffusion of the difference coming from the message words, any characteristic that does not follow this strategy will have a low probability.
- In the backward direction, the differences in the message words are chosen as follows: $\Delta w_2 = [+31]$, $\Delta w_i = 0$ $(0 \leq i \leq 15, i \neq 2)$. Because Δw_2= [+31] and $w_2' = w_2 \oplus w_6$, we choose $w_{6,31} = 0$ to ensure that $\Delta w_2'$= [+31]. Since $w_{18} = P_1(w_2 \oplus w_9 \oplus (w_{15} \lll 15)) \oplus (w_5 \lll 7) \oplus w_{12}$, by choosing proper $w_{12,14}, w_{12,22}$ and $w_{12,31}$, we can ensure that Δw_{18}= [+14, +22, +31] holds. Combined with $w_{14,i} = 0$

Table 2. Differential characteristic for steps 1-16 using signed bit-wise differences (32 steps)

Step	Differences of chaining values	w	w'	Pr	Sufficient conditions
	B_0:[+22] C_0:[-31] D_0:[-22,+31] F_0:[+12] G_0:[-31] H_0:[-12,+31]			2^{-6}	$A_{0,22} = C_{0,22}$, $E_{0,12} = G_{0,12}$, $D_{0,22} = 1, D_{0,31} = 0$, $H_{0,i} = 1 (i = 12, 31)$,
1	C_1:[+31] D_1:[-31] G_1:[+31] H_1:[-31]			2^{-2}	$D_{1,31} = 1$, $H_{1,31} = 1$.
2	D_2:[+31] H_2:[+31]	[+31]	[+31]	2^{-2}	$D_{2,31} = 0$, $H_{2,31} = 0$.
3				1	
⋮	⋮	⋮	⋮	⋮	⋮
14			[+14,+22,+31]	1	
15	A_{15}:[+14,+22,+31]			2^{-57}	$A_{16,i} = 0 (i = 1, 2, 9, 11, 14, 18, 22, 26, 31)$, $B_{16,i} = 0 (i = 6, 14, 21, 22, 29, 31)$, $C_{16,8} = D_{16,31}, C_{16,23} = D_{16,14}$, $C_{16,31} = D_{16,22}$, $(B_{16} \lll 12) + F_{16} = -(T_{15} \lll 15)$, $E_{16,i} = 0 (i = 1, 3, 9, 10, 18, 26, 27)$, $\bigoplus_{i \in \Lambda_j} E_{16,i} = 0,\ j \in \{1, 2, 3\}$, $\Lambda_1 = \{6, 14, 15, 16, 22, 24, 30, 31\}$, $\Lambda_2 = \{0, 6, 7, 14, 22, 23, 24, 30\}$, $\Lambda_3 = \{0, 7, 15, 16, 23, 31\}$.
16	A_{16}:[+1,+2,+9,+11, +14,+18,+22, +26,+31] B_{16}:[+14,+22,+31] E_{16}:[+1,+3,+9,+10 +18, +26, +27]				

$(i = 14, 22, 31)$, we can get $\Delta w_{14}'$= [+14, +22, +31]. So the differences of the message words in the backward direction are Δw_2 = [+31], $\Delta w_2'$ = [+31], $\Delta w_{14}'$ = [+14, +22, +31], and all the other message words differences are zero.

If A_{15}[+14, +22, +31] holds, then we can cancel the differences $\Delta w_{14}'$ = [+14, +22, +31] in step 15, and skip 12 steps from step 15 to step 4 with probability 1. The following is the derivation for the sufficient conditions in step 16 of Table 2. The differential characteristic in step 16 is given by:

Table 3. Differential characteristic for steps 17-32 using signed bit-wise differences

Step	Differences of chaining values	w	w'	Pr	Sufficient conditions
16	A_{16}:[-6] D_{16}:[-6,-11,-13, -18,-19,-20, -22,-25,-26] E_{16}:[-28] H_{16}:[-4,-5,-11, -13,-19,-20, -22,-25,-26]	[+3,+4,+5, +11,+13, +19,+20, +22,+27]	[+3,+4,+5, +11,+13, +19,+20, +22,+27]		
17	B_{17}:[-6] F_{17}:[-28]			2^{-27}	$A_{16,i} = 1(i = 6, 23), A_{16,13} = 0,$ $B_{16,6} = C_{16,6},$ $D_{16,i} = 1(i = 6, 11, 13, 18, 19, 20, 22, 25, 26),$ $E_{16,28} = 1, F_{16,28} = G_{16,28},$ $H_{16,i} = 1(i = 4, 5, 11, 13, 19, 20, 22, 25, 26).$ $SS1_{16,i} = 1(i = 3, 18, 25).$
18	C_{18}:[-15] G_{18}:[-15]			2^{-2}	$A_{17,6} = C_{17,6},$ $E_{17,28} = 0.$
19	D_{19}:[-15] H_{19}:[-15]	[+15]	[+15]	2^{-2}	$A_{18,15} = B_{18,15},$ $E_{18,15} = 1.$
20				1	
⋮	⋮	⋮	⋮	⋮	⋮
31			[±6, +15, +30][1]	1	
32	A_{32}:[±6,+15,+30]			2^{-3}	

[1] The difference of w_2 affects $w_{31,6}$. In other words, if we choose $w^X_{31,6} - w^Y_{31,6} = +1$, then the difference $w^{X'}_{31,6} - w^{Y'}_{31,6} = -1$, and vice versa (We used " ± " to denote this fact). Note that this does not affect the XOR-differences in step-32.

$$(A_{16}[+1, +2, +9, +11, +14, +18, +22, +26, +31], B_{16}[+14, +22, +31], C_{16}, D_{16},$$

$$E_{16}[+1, +3, +9, +10, +18, +26, +27], F_{16}, G_{16}, H_{16}) \longrightarrow$$

$$(A_{15}[+14, +22, +31], B_{15}, C_{15}, D_{15}, E_{15}, F_{15}, G_{15}, H_{15}).$$

1. Because $A_{15} = B_{16}, B_{15} = (C_{16} \ggg 9), C_{15} = D_{16}$, $E_{15} = F_{16}, F_{15} = (G_{16} \ggg 19)$ and $G_{15} = H_{16}$, we choose $(B_{16} \lll 12)+F_{16} = -(T_{15} \lll 15)$ to ensure that $\Delta SS1 = \Delta((B_{16} \lll 12)+F_{16} + (T_{15} \lll 15)) \lll 7 = [+1, +9, +18]$ holds and the conditions $B_{16,i} = 0(i = 6, 21, 29)$ ensure that $\Delta SS2 = \Delta SS1 \oplus \Delta(B_{16} \lll 12)= [+1, +2, +9, +11, +18, +26]$ holds.
2. The conditions $B_{15,i} = C_{15,i}$, $(i = 14, 22, 31)$, i.e., $C_{16,8}{=}D_{16,31}$, $C_{16,23}{=}D_{16,14}$ and $C_{16,31}{=}D_{16,22}$ ensure that $\Delta FF_{15}(A_{15}, B_{15}, C_{15})= \Delta FF_{15}(B_{16}, C_{16} \ggg 9, F_{16})= [+14, +22, +31]$ hold. Combined with $\Delta w_{15}' = 0$, we can get $\Delta D_{15} = \Delta A_{16}$ - $(\Delta FF_{15}(B_{16}, C_{16} \ggg 9, D_{16}) + \Delta SS2 + \Delta w_{15}') = 0$. Similarly, the conditions $\bigoplus_{i\in\Lambda_j} E_{16,i} = 0$, $j \in \{1, 2, 3\}$, $\Lambda_1 = \{6, 14, 15, 16, 22, 24, 30, 31\}$, $\Lambda_2 = \{0, 6, 7, 14, 22, 23, 24, 30\}, \Lambda_3 = \{0, 7, 15, 16, 23, 31\}$ ensure that $\Delta H_{15} = 0$ holds.

Thus the above conditions constitute a set of sufficient conditions for the differential characteristic in step 16.

Table 4. Differential characteristic for steps 1-16 using signed bit-wise differences (33 steps)

Step	Differences of chaining values	w	w'	Pr	Sufficient conditions
	B_0:[+22] C_0:[-31] D_0:[-22,+31] F_0:[+12] G_0:[-31] H_0:[-12,+31]			2^{-6}	$A_{0,22} = C_{0,22}$, $E_{0,12} = G_{0,12}$, $D_{0,22} = 1, D_{0,31} = 0$, $H_{0,i} = 1 (i = 12, 31)$.
1	C_1:[+31] D_1:[-31] G_1:[+31] H_1:[-31]			2^{-2}	$D_{1,31} = 1$, $H_{1,31} = 1$.
2	D_2:[+31] H_2:[+31]	[+31]	[+31]	2^{-2}	$D_{2,31} = 0$, $H_{2,31} = 0$.
3				1	
⋮	⋮	⋮	⋮	⋮	⋮
14			[+14,+22,+31]	1	
15	A_{15}:[+14,+22,+31]			2^{-59}	$A_{16,i} = 0 (i = 3, 9, 11, 14, 15, 16, 17, 22, 26, 31)$, $A_{16,1} = 1, B_{16,i} = 0 (i = 6, 14, 21, 22, 29, 31)$, $C_{16,8} = D_{16,31}, C_{16,23} \neq D_{16,14}, C_{16,31} = D_{16,22}$, $(B_{16} \lll 12) + F_{16} = -(T_{15} \lll 15)$, $E_{16,i} = 0 (i = 1, 3, 9, 10, 18, 26, 27)$, $\bigoplus_{i \in \Lambda_j} E_{16,i} = 0, j \in \{1, 2, 3\}$, $\Lambda_1 = \{6, 14, 15, 16, 22, 24, 30, 31\}$, $\Lambda_2 = \{0, 6, 7, 14, 22, 23, 24, 30\}$, $\Lambda_3 = \{0, 7, 15, 16, 23, 31\}$.
16	A_{16}:[-1,+3,+9,+11, +14,+15,+16, +17,+22, +26,+31] B_{16}:[+14,+22,+31] E_{16}:[+1,+3,+9,+10 +18, +26, +27]				

– In the forward direction, we choose the message differences as follows: $\Delta w_{19} = [+15]$, $\Delta w_i = 0 (20 \leq i \leq 34)$. Since $w_{19}' = w_{19} \oplus w_{23}$, we choose $w_{23,15} = 0$ to ensure that $\Delta w_{19}' = [+15]$. Because $\Delta w_{19} = [+15]$ and $w_{19} = P_1(w_3 \oplus w_{10} \oplus (w_{16} \lll 15)) \oplus (w_6 \lll 7) \oplus w_{13}$, let $w_3 \oplus w_{10} = 0$ and $(w_6 \lll 7) \oplus w_{13} = 0$, to get Δw_{16}= [+3, +4, +5, +11, +13, +19, +20, +22, +27]. If we choose $w_{20,i} = 0$, $(i = 3, 4, 5, 11, 13, 19, 20, 22, 27)$ and $w_{23,15} = 0$, from $w_{16}' = w_{16} \oplus w_{20}$, we can get $\Delta w_{16}' = \Delta w_{16}$. Because $w_{35} = P_1(w_{19} \oplus w_{26} \oplus (w_{32} \lll 15)) \oplus (w_{22} \lll 7) \oplus w_{29}$ and $w_{31}' = w_{31} \oplus w_{35}$, we choose $w_{31,i} = 0$ $(i = 6, 15, 30)$, such that $\Delta w_{31}' = [\pm 6, +15, +30]$[1].

[1] Because $w_{31} = P_1(w_{15} \oplus w_{22} \oplus (w_{28} \lll 15)) \oplus (w_{18} \lll 7) \oplus w_{25}$ and Δw_{18}= [+14, +22, +31] in the backward direction, if we choose $w^X_{31,i} = 0$ and $w^Y_{31,i} = 1 (i = 6, 15, 30)$, then the bits in $w^{X'}_{31}$ and $w^{Y'}_{31}$ are $w^{X'}_{31,i} = 0 (i = 15, 30)$, $w^{X'}_{31,6} = 1$ and $w^{Y'}_{31,i} = 1 (i = 15, 30)$, $w^{Y'}_{31,6} = 0$. So

Table 5. Differential characteristic for steps 17-33 using signed bit-wise differences

Step	Differences of chaining values	w	w'	Pr	Sufficient conditions
16	C_{16}:[-0,-2,-5,-6,-18, -23,-25,-30,-31] D_{16}:[-27] G_{16}:[-0,-2,-5,-6,-16, -17,-23,-25,-31] H_{16}:[-3,-8,-10,-11, -19,-26,-27]		[+27]		
17	A_{17}:[-18] D_{17}:[-0,-2,-5,-6,-18, -23,-25,-30,-31] E_{17}:[-8] H_{17}:[-0,-2,-5,-6,-16, -17,-23,-25,-31]	[+0,+2,+7, +15,+16, +17,+23, +25,+31]	[+0,+2,+7, +15,+16, +17,+23, +25,+31]	2^{-54}	$A_{16,i}{=}B_{16,i}(i = 0, 2, 5, 6, 23, 25, 30, 31)$, $A_{16,18} \neq B_{16,18}$, $C_{16,i} = 1(i = 0, 2, 5, 6, 18, 23, 25, 30, 31)$, $TT1_{16,8} = 0$,$D_{16,27} = 1$, $E_{16,i}{=}1(i = 0, 2, 5, 6, 16, 23, 31)$, $E_{16,i} = 0(i = 17, 25)$, $G_{16,i} = 1(i = 0, 2, 5, 6, 16, 17, 23, 25, 31)$, $H_{16,i} = 1(i = 3, 8, 10, 11, 19, 26, 27)$, $TT2_{16,i} = 1(i = 3, 8, 10, 11, 17, 19, 25, 26, 27)$.
18	B_{18}:[-18] F_{18}:[-8]			2^{-9}	$A_{17,i} = 1(i = 3, 18), A_{17,25} = 0$, $B_{17,18} = C_{17,18}$, $E_{17,8} = 1, F_{17,8} = G_{17,8}$, $SS1_{17,i} = 1(i = 5, 15, 30)$.
19	C_{19}:[-27] G_{19}:[-27]			2^{-2}	$A_{18,18} = C_{18,18}$, $E_{18,8} = 0$.
20	D_{20}:[-27] H_{20}:[-27]	[+27]	[+27]	2^{-2}	$A_{19,27} = B_{19,27}$, $E_{19,27} = 1$.
21				1	
⋮	⋮	⋮	⋮	⋮	⋮
32			[+10,+18,+27]	1	
33	A_{32}:[+10,+18,+27]			2^{-3}	

In this case, the massage word differences in the forward direction are $\Delta w_{16} = \Delta w_{16}' = [+3, +4, +5, +11, +13, +19, +20, +22, +27], \Delta w_{19} = \Delta w_{19}' = [+15], \Delta w_{31}' = [\pm 6, +15, +30]$, and all the other message words differences are zero.

The following is the derivation for the sufficient conditions for step 17 in Table 3. The differential characteristic in step 17 is given by:

$$(A_{16}[-6], B_{16}, C_{16}, D_{16}[-6, -11, -13, -18, -19, -20, -22, -25, -26], E_{16}[-28], F_{16},$$

$$G_{16}[+28], H_{16}[-4, -5, -11, -13, -19, -20, -22, -25, -26]) \longrightarrow$$

$$(A_{17}, B_{17}[-6], C_{17}, D_{17}, E_{17}, F_{17}[-28], G_{17}, H_{17}).$$

the difference of w_2 affects $w_{31,6}$. In other words, if we choose $w^X_{31,6} - w^Y_{31,6} = +1$, then the difference $w^{X'}_{31,6} - w^{Y'}_{31,6} = -1$, and vice versa (We used " ± " to denote this fact). Note that this does not affect the XOR-differences in step 32.

1. From $\Delta A_{16} = [-6]$ and $B_{16,6} = C_{16,6}$, we get $\Delta FF_{16}(A_{16}, B_{16}, C_{16}) = 0$. From $\Delta E_{16} = [-28]$ and $F_{16,28} = G_{16,28}$, we get $\Delta GG_{16}(E_{16}, F_{16}, G_{16}) = 0$.
2. From $\Delta A_{16} = [-6]$ and $\Delta E_{16} = [-28]$, it follows that the conditions $SS1_{16,3} = 1, SS1_{16,25} = 1$ ensure $\Delta SS1_{16} = [-3, -25]$. Combined with the conditions $SS1_{16,18} = 1, A_{16,13} = 0$ and $A_{16,23} = 1$, we can get $\Delta SS2_{16} = [+3, +18, -25]$. Therefore, $\Delta A_{17} = \Delta TT1_{16} = 0$, and $\Delta E_{17} = \Delta P_0(TT2_{16}) = 0$.

Thus the above conditions constitute a set of sufficient conditions for the differential characteristic in step 17.

4.2 Message Modification

We use the message modification technique which has been introduced by Wang *et al.* [18] to improve the complexity of our attack. We can modify the chaining values A_{16} to H_{16} to ensure that almost all the conditions in A_i to H_i (i=17,18,19) hold. For example, in the backward direction, we can modify $E_{16,i}(i = 6, 15)$ to make the sufficient conditions $\bigoplus_{i\in\Lambda_j} E_{16,i} = 0$, $j\in \{1, 2, 3\}$, $\Lambda_1 = \{6, 14, 15, 16, 22, 24, 30, 31\}$, $\Lambda_2 = \{0, 6, 7, 14, 22, 23, 24, 30\}$, $\Lambda_3 = \{0, 7, 15, 16, 23, 31\}$ hold.

In Table 3, there are 31 sufficient conditions from step 17 to step 19 in each differential. We can correct all the sufficient conditions in one differential by using message modification techniques, and the sufficient conditions $SS1_{16,i} = 1(i = 3, 18, 25)$, $A_{17,6} = C_{17,6}$, $E_{17,28} = 0$, $A_{18,15} = B_{18,15}$ and $E_{18,15} = 1$ are not corrected in another differential. So the probability of step 17 to step 19 can be improved from $2^{-2\times31} = 2^{-62}$ to 2^{-7}. In Table 5, there are 63 sufficient conditions from step 17 to step 18 in each differential. We can correct all the sufficient conditions in one differential by using message modification techniques. However, the sufficient conditions $TT1_{16,8} = 0$, $TT2_{16,i} = 1(i = 3, 8, 10, 11, 17, 19, 25, 26, 27), A_{17,i} = 1(i = 3, 18), A_{17,25} = 0, E_{17,8} = 1$ and $SS1_{17,i} = 1(i = 5, 15, 30)$ are not corrected in the other differential. So the probability of step 17 to step 18 can be improved from $2^{-2\times63} = 2^{-126}$ to 2^{-17}. Consequently, in this case, the probability of step 17 to step 20 can be improved from $2^{-2\times67} = 2^{-134}$ to $2^{-17-2\times4} = 2^{-25}$.

4.3 Boomerang Attacks on the 32-Step SM3 Compression Function

The attack algorithm on 32-step SM3 compression function can be summarized as follows.

1. Choose a random 512-bit message M and expand it to 36 words. Set proper message words as in section 4.1 to ensure that $\Delta w_2 = [+31]$, $\Delta w_2{}' = [+31]$, $\Delta w_{14}{}' = [+14, +22, +31]$ in the backward direction, and $\Delta w_{16} = \Delta w_{16}{}' = [+3, +4, +5, +11, +13, +19, +20, +22, +27]$, $\Delta w_{19} = [+15]$, $\Delta w_{19}{}' = [+15]$, $\Delta w_{31}{}' = [\pm6, +21, +29]$ in the forward direction. Let $M_X = M$, $M_{X'} = M \oplus \Delta w_2$. Expend the messages M_X and $M_{X'}$ to 36 words W_X and $W_{X'}$, respectively. Let $W_Y = W_X \oplus \Delta w_{19}$ and $W_{Y'} = W_{X'} \oplus \Delta w_{19}$. Then we use the 16 words $(W_Y)_{19}^{34}$ and $(W_{Y'})_{19}^{34}$ to get two 36-word $(W_Y)_0^{35}$ and $(W_{Y'})_0^{35}$ by using the message expansion algorithm. Let $M_Y = (W_Y)_0^{15}$, $M_{Y'} = (W_{Y'})_0^{15}$.

Table 6. Example of a quartet for 32 steps of the SM3 compression function

Message								
M_X	ecffec51	192fa6b4	6314d06c	f86f5604	ed6f140d	4597860c	3a4ccc9a	b78b2ded
	6a5b06f8	33169484	355b11c5	6b81ddd0	1e58820e	78fa46f6	742b217f	4669940f
$M_{X'}$	ecffec51	192fa6b4	e314d06c	f86f5604	ed6f140d	4597860c	3a4ccc9a	b78b2ded
	6a5b06f8	33169484	355b11c5	6b81ddd0	1e58820e	78fa46f6	742b217f	4669940f
M_Y	e8fff051	1bafa634	6b4cf854	a86ff654	6dea968e	4597060c	524cc49a	b78b25ed
	6a5b06f8	430624d4	3d5311d5	6b81ddd0	1e58020e	50fa4ef6	742b217f	4669940f
$M_{Y'}$	e8fff051	1bafa634	eb4cf854	a86ff654	6dea968e	4597060c	524cc49a	b78b25ed
	6a5b06f8	430624d4	3d5311d5	6b81ddd0	1e58020e	50fa4ef6	742b217f	4669940f
Chaining Value								
IV_X	0d548434	6f039a92	3d5fb868	01b03347	29c6a571	0d8b6217	4f2359fa	d6a363f4
$IV_{X'}$	0d548434	6f439a92	bd5fb868	81703347	29c6a571	0d8b7217	cf2359fa	56a373f4
IV_Y	792457dc	8f057732	6137fcd4	7899b663	948b29bf	d5f5a832	d9ae3751	c747e405
$IV_{Y'}$	792457dc	8f457732	e137fcd4	f859b663	948b29bf	d5f5b832	59ae3751	4747f405
H_X	a1e82b03	54b1bb42	2563b063	e514d921	a0eaf1fa	632d0eef	e2a999cf	ad4964d1
$H_{X'}$	3fd356c7	ca7f9b81	3a9694d6	31d02769	b454a3bd	c2d2dc37	45ffc720	2e319c71
H_Y	e1e8ab43	54b1bb42	2563b063	e514d921	a0eaf1fa	632d0eef	e2a999cf	ad4964d1
$H_{Y'}$	7fd3d687	ca7f9b81	3a9694d6	31d02769	b454a3bd	c2d2dc37	45ffc720	2e319c71

2. Randomly choose the chaining values A_{16}, B_{16}, C_{16}, D_{16}, E_{16}, G_{16} and H_{16} such that almost all the conditions used in step 16 and step 17[2] in Table 2 and Table 3 hold.
3. By using the message modification technique, modify $A_{16,19}$, $H_{16,28}$, $C_{16,15}$ and $G_{16,15}$ to make the sufficient conditions $A_{17,6} = C_{17,6}$, $E_{17,28} = 0$, $A_{18,15} = B_{18,15}$ and $E_{18,15} = 1$ hold in one of the differentials in the forward direction.
4. Use state update transformation process to get IV_X, IV_Y, $IV_{X'}$, $IV_{Y'}$, H_X, H_Y, $H_{X'}$ and $H_{Y'}$. Check whether $IV_X \oplus IV_{X'} = IV_Y \oplus IV_{Y'}$ and $H_X \oplus H_Y = H_{X'} \oplus H_{Y'}$ hold.
5. If a quartet is found, then a distinguisher is found. Repeat the above 4 steps with different messages and chaining values (A_{16} to H_{16}) until a distinguisher is found.

4.4 Complexity of the Attack

Using the differential characteristics and the message modification technique, we can construct the boomerang attack for the SM3 compression function reduced to 32 steps.

In the backward direction, all the sufficient conditions used in step 16 can be set in both of the differentials and the sufficient conditions used in step 3 to step 1 cannot be corrected in both of the differentials. So the differential characteristic used in the backward direction holds with probability 2^{-10}. Thus both of the differentials used in the backward direction hold with probability $2^{-10\times 2} = 2^{-20}$. In the forward direction, 7 sufficient conditions, from step 17 to step 19, are not corrected in one of the differentials by using message modifications and in step 32, non of the sufficient conditions is corrected in both of the differentials. Thus both of the differentials used in the forward direction hold with probability $2^{-7-3\times 2} = 2^{-13}$ after the message modification.

Hence, we can give a boomerang attack on 32-step SM3 with complexity $2^{20+13} = 2^{33}$. We can also use the amplified differential characteristics to improve the complexity of the attack. In this case, both of the two differentials used in the backward direction hold with probability $2^{-3.2}$ and the two differentials used in step 32 hold with probability

[2] All the conditions used in step 16 can hold in both of the differentials. In step 17 the conditions $SS1_{16,i} = 1 (i = 3, 18, 25)$ cannot be corrected in one of the differentials and all the other conditions can hold.

$2^{-4.2}$. So we can get a 32-step boomerang distinguisher with complexity $2^{3.2+7+4.2} = 2^{14.4}$. An example of a 32-step boomerang distinguisher (quartet) is presented in Table 6.

4.5 Attacks on 33/34/35 Steps SM3 Compression Function

In what follows we extend the proposed boomerang distinguisher to the 35-step SM3 compression function. First, we obtain a new 33-step boomerang distinguisher and then extend it to 35 steps. If we simply add one step in the forward differential characteristic in Table 3, this will result in some contradictions in A_{16} and E_{16} between Table 2 and Table 3. So we choose the message words differences as follows: $\Delta w_2 = [+31]$, $\Delta w_i = 0$ $(0 \leq i \leq 15, i \neq 2)$ in the backward direction, and $\Delta w_{20} = [+27]$, $\Delta w_i = 0 (21 \leq i \leq 35)$ in the forward direction.

We also correct $\Delta A_{16} = [-1, +3, +9, +11, +14, +15, +16, +17, +22, +26, +31]$ and change one of the sufficient conditions $C_{16,23} = D_{16,14}$ to $C_{16,23} \neq D_{16,14}$.

In this case, the backward direction is from step 16 to step 1 and the differential characteristic which is given in Table 4 holds with probability $2^{-3.2}$. The forward direction is from step 17 to step 33 and the differential characteristic is given in Table 5 where 25 sufficient conditions are not corrected in one of the differentials by using message modifications. So the forward differential characteristic holds with probability $2^{-25-4.2} = 2^{-29.2}$. So we can get the 33-step boomerang distinguisher with complexity $2^{3.2+29.2} = 2^{32.4}$. In step 34 both of the differentials hold with probability $2^{-20.7}$ using the amplified differential characteristics. Thus we obtain a 34-step boomerang distinguisher with complexity $2^{32.4+20.7} = 2^{53.1}$. We also assume A_{35} and E_{35} all have 32-bit differences. Thus the 35-step boomerang distinguisher has a complexity $\approx 2^{53.1+2\times 32} = 2^{117.1}$.

5 A Slide-Rotational Property of SM3-XOR

In this section, we show that, in the case of the full SM3-XOR, pairs satisfying a certain rotational relation can be easily generated. An example of such a pair for the SM3-XOR is provided in Table 8. Such a property is not known to exist for SHA2-XOR [17].

The above mentioned property exists due to the fact that the constants over the 64 steps of SM3 are related. According to the SM3 specification, in steps $j \in \{0, \ldots, 15\}$, one constant rotated by j is utilized, whereas the other constant rotated by j is used in steps $j \in \{16, \ldots, 63\}$. Since operations like XOR, FF_i, GG_i, $0 \leq i < 64$, that are used in the SM3-XOR step function preserve the rotational property, it is natural to attempt a rotational attack, as provided below. We note that if instead of SM3-XOR, the original SM3 compression function is used, the addition mod 2^{32} transforms the attack into a probabilistic one, as outlined below. Due to the high number of additions per step, it appears difficult to exploit this rotational property directly and therefore the security of the SM3 compression function, at this stage of analysis, does not seem to be directly affected.

Two 32-bit words X, Y are said to be *rotational* if $X = Y \lll n$. Let messages W and W^* satisfy $W_1^* = W_0 \lll 1, W_2^* = W_1 \lll 1, \ldots, W_{16}^* = W_{15} \lll 1$. Below, a procedure for the instant generation of pairs v, v^* such that

Table 7. Example of a quartet for 33 steps of the SM3 compression function

	Message							
M_X	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
$M_{X'}$	00000000	00000000	80000000	00000000	00000000	00000000	00000000	00000000
	00000000	00000000	00000000	00000000	00000000	00000000	00000000	00000000
M_Y	04001c00	02800080	08582838	5000a050	80858283	00008000	68000800	00000800
	00000000	7010b050	08080010	00000000	00008000	28000800	00000000	00000000
$M_{Y'}$	04001c00	02800080	88582838	5000a050	80858283	00008000	68000800	00000800
	00000000	7010b050	08080010	00000000	00008000	28000800	00000000	00000000
	Chaining Value							
IV_X	274e6355	3333edb0	14f1b3d9	7be58154	d969d138	bb60c21a	ff5909df	e92dce5d
$IV_{X'}$	274e6355	3373edb0	94f1b3d9	fba58154	d969d138	bb60d21a	7f5909df	692dde5d
IV_Y	28b7b4d8	fe5f1155	93973138	c10d3808	32d4319b	dc8de94e	ef594319	8ef80fe1
$IV_{Y'}$	28b7b4d8	fe1f1155	13973138	414d3808	32d4319b	dc8df94e	6f594319	0ef81fe1
H_X	52793642	8017615c	fbf548ba	8b05cf67	dcb79a73	e1035e10	2caefeae	701d22d9
$H_{X'}$	772427a1	b2064c80	0dd79a89	2a809122	8bc2413f	8dd6b954	bad8867b	06c59c18
H_Y	987f3286	c017e19c	fbf548ba	8b05cf67	dabd9677	e1035e10	2caefeae	701d22d9
$H_{Y'}$	bd222365	f206cc40	0dd79a89	2a809122	8dc84d3b	8dd6b954	bad8867b	06c59c18

Table 8. An example for a slide-rotational pair for the SM3-XOR compression function

$A^1, B^1, \ldots, H^1$	0x565060b7 0x125d5655 0x285c7653 0xeaf5fe1e 0xda8bd7dd 0xb8bb1904 0x43bcaf18 0x7cf88895
$W_0^1, \ldots, W_{15}^1$	0x8f450bbd 0x4a0c9922 0x73dd44f8 0x9eceaaf8 0x33b13e20 0xb59d9c33 0x6b5a5f23 0xc0d2b468 0x7a9a1e16 0xaff62878 0x3fbb01f4 0x75278787 0xac0b849e 0x498f3045 0x62687c15 0xd3498eb
$A^2, B^2, \ldots, H^2$	0x24baacaa 0x53285c76 0xd5ebfc3d 0xdf1ee2a6 0x71763209 0x2bc610ef 0xf9f1112a 0xffeb86a4
$W_0^2, \ldots, W_{15}^2$	0x7efa7542 0x1e8a177b 0x94193244 0xe7ba89f0 0x3d9d55f1 0x67627c40 0x6b3b3867 0xd6b4be46 0x81a568d1 0xf5343c2c 0x5fec50f1 0x7f7603e8 0xea4f0f0e 0x5817093d 0x931e608a 0xc4d0f82a

$$\begin{aligned} v_1^* &= v_0 \lll 1, v_2^* = v_1 \lll 8, v_3^* = v_2 \lll 1 \\ v_5^* &= v_4 \lll 1, v_6^* = v_5 \lll 18, v_7^* = v_6 \lll 1 \\ V_1^* &= V_0 \lll 1, V_2^* = V_1 \lll 8, V_3^* = V_2 \lll 1 \\ V_5^* &= V_4 \lll 1, V_6^* = V_5 \lll 18, V_7^* = V_6 \lll 1 \end{aligned} \tag{4}$$

is provided, where V = SM3-XOR(v, W), V^* = SM3-XOR(v^*, W^*) and v_i, V_i for $0 \leq i \leq 7$ denote i-th 32-bit word in the v and V, respectively. For a random function, a random (v, W), (v^*, W^*) satisfying the above constraints will yield the corresponding V and V^* with probability $2^{-6\times32} = 2^{-192}$, since (4) imposes 6 32-bit conditions on V, V^*.

5.1 Constructing a Slide-Rotational Pair

In this section, step i denotes the transformation from $(A_i, B_i, C_i, D_i, E_i, F_i, G_i, H_i)$ to $(A_{i+1}, B_{i+1}, C_{i+1}, D_{i+1}, E_{i+1}, F_{i+1}, G_{i+1}, H_{i+1})$. For example by step 0 the first compression function step is denoted. We start by the following observations:

- The slide rotational messages expand to slide-rotational expanded messages with probability 1. In particular, fix $W_0, \ldots, W_{15}$ and let

$$W_1^* = W_0 \lll 1, W_2^* = W_1 \lll 1, \ldots, W_{16}^* = W_{15} \lll 1 \tag{5}$$

  After expanding both W and W^*, we have $W_{i+1}^* = W_i \lll 1$, for $i = \{0, 1, \ldots, 62\}$ and also $W'^*_{i+1} = W'_i \lll 1$, for $i = \{0, 1, \ldots, 66\}$.

- We recall that T_i, $0 \le i \le 63$ are the step constants. If we have

$$W^*_{i+1} = W_i \lll 1, W'^*_{i+1} = W'_i \lll 1, T_{i+1} = T_i \lll 1 \quad (6)$$

$$A^*_{i+1} = A_i \lll 1, B^*_{i+1} = B_i \lll 1, \ldots, H^*_{i+1} = H_i \lll 1 \quad (7)$$

for $i = k$, then (7) will also hold for $i = k + 1$, where $k = 0, \ldots, 62$.

The observations above suggest that sliding can be introduced, as depicted in Fig. 2.

Namely, consider randomly initializing W and letting W^* satisfy (5). Moreover, $A_0, B_0 \ldots, H_0$ is chosen randomly and the inner state registers after the first step in the second instance of the hash function are initialized according to (7). Then, until step 15, due to (6), the rotational property in the inner state registers will be preserved. Once the two instances reach steps 15 and 16, respectively, a different step transformation is applied in the two instances and the rotational property may discontinue. This problem is bypassed by starting from the middle, i.e., by populating the inner states entering the critical steps 15 and 16 (see Fig. 2).

5.2 Bypassing Steps 15 and 16

The idea is to start by populating the inner states entering the critical steps 15 and 16 (see Fig. 2). In particular, a rotational pair $(A_{15}, \ldots, H_{15})$, $(A^*_{16}, \ldots, H^*_{16})$ is carefully chosen so that $(A_{16}, \ldots, H_{16})$ and $(A^*_{17}, \ldots, H^*_{17})$ satisfy relation (7). It should be noted that the rotational property may be destroyed only between A_{16} and A^*_{17} and between E_{16} and E^*_{17}, since the other registers go through identical rotational-preserving transformations in step 15 and step 16. As for A_{16} and A^*_{17}, for the purpose of tracking the possible rotational disturbance between the two registers, the equation to compute these two registers can be rewritten as

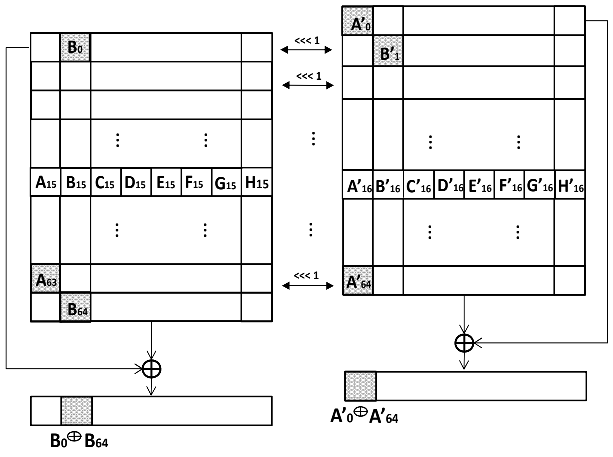


Fig. 2. The slide-rotational attack against SM3-XOR

$$A_{16} = FF_{15}(A_{15}, B_{15}, C_{15}) \oplus (T_{15} \lll 22) \oplus \alpha \tag{8}$$

$$A^*_{17} = FF_{16}(A^*_{16}, B^*_{16}, C^*_{16}) \oplus (T_{16} \lll 23) \oplus \alpha^* \tag{9}$$

where $\alpha = D_{15} \oplus W_{15} \oplus W_{19} \oplus (((A_{15} \lll 12) \oplus E_{15}) \lll 7) \oplus (A_{15} \lll 12)$ and $\alpha^* = D^*_{16} \oplus W^*_{16} \oplus W^*_{20} \oplus (((A^*_{16} \lll 12) \oplus E^*_{16}) \lll 7) \oplus (A^*_{16} \lll 12)$. Since (7) and (6) hold for $i = 15$, $\alpha^* = \alpha \lll 1$. Therefore, to have A_{16} and A^*_{17} be a rotational pair, it suffices to make $FF_{15}(A_{15}, B_{15}, C_{15}) \oplus (T_{15} \lll 22)$ and $FF_{16}(A^*_{16}, B^*_{16}, C^*_{16}) \oplus (T_{16} \lll 23)$ satisfy the rotational property. After expressing A^*_{16}, B^*_{16}, C^*_{16} in terms of A_{15}, B_{15}, C_{15} and using that FF_{15} and FF_{16} preserve the rotational property, the condition can be expressed in terms of A_{15}, B_{15}, C_{15} as follows:

$$FF_{15}(A_{15}, B_{15}, C_{15}) \oplus FF_{16}(A_{15}, B_{15}, C_{15}) = (T_{15} \oplus T_{16}) \lll 22 \tag{10}$$

When applied on 1-bit values X, Y and Z, the equation $FF_{15}(X, Y, Z) \oplus FF_{16}(X, Y, Z) = 0$ is satisfied for 2 out of 8 (X, Y, Z) values. Since the Hamming weight of the right-hand side of (10) is equal to 14, the number of solutions to the equation is $2^{18} \times 6^{14} = 2^{32} \times 3^{14}$. As for preserving the rotational property between E_{16} and E^*_{17}, developing the registers as in (8) and then forming the equation of the form (10) yields that the number of solutions E_{15}, F_{15} and G_{15} is $4^{32} = 2^{64}$. Therefore, the number of solutions for $(A_{15}, \ldots, H_{15})$ that pass the disturbance in steps 15 and 16 is $2^{32} \times 3^{14} \times 2^{64} \times 2^{64} \approx 2^{182.19}$, since D_{15} and H_{15} are free variables. For such pairs, it follows that relations (4) are satisfied.

When instead of SM3-XOR, the SM3 compression function is considered, this property turns into a probabilistic one. Following [15], if $p_r = P[(x \lll r) + (y \lll r) = (x + y) \lll r]$ where x and y are 32-bit words, then $p_1 = 2^{-1.415}$. Since there exists 8 additions in one SM3 step, the probability that one step and its corresponding slided step will preserve the rotational property is given by $(p_1)^8 = 2^{-11.320}$ [15].

6 Conclusions

In this paper, we have shown an application of the boomerang-style attack on the step-reduced SM3 compression function. In particular, we presented distinguishing attacks for 32 steps of the compression function with complexity $2^{14.4}$, 33 steps with complexity $2^{32.4}$, 34 steps with complexity $2^{53.1}$ and 35 steps with complexity $2^{117.1}$. Our results suggest that 35-step SM3 compression does not behave randomly. In the second part of the paper, a slide-rotational property of SM3-XOR function is exposed and an example of a slide-rotational pair for SM3-XOR compression function is given.

Acknowledgment. The authors would like to thank the anonymous reviewers for valuable comments that helped improve the presentation of the paper. The work of Gaoli Wang and Yanzhao Shen is supported by the National Natural Science Foundation of China (No. 61103238), Chen Guang project (supported by Shanghai Municipal Education Commission and Shanghai Education Development Foundation), the Fundamental Research Funds for the Central Universities. The work of Aleksandar Kircanski and Amr Youssef is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

1. Specification of SM3 cryptographic hash function (in Chinese), http://www.oscca.gov.cn/UpFile/20101222141857786.pdf/
2. Zou, J., Wu, W., Wu, S., Su, B., Dong, L.: Preimage Attacks on Step-Reduced SM3 Hash Function. In: Kim, H. (ed.) ICISC 2011. LNCS, vol. 7259, pp. 375–390. Springer, Heidelberg (2012)
3. Aumasson, J.P.: Zero-sum Distinguishers. Rump session talk at CHES 2009 (2009), http://131002.net/data/papers/AM09.pdf
4. Wagner, D.: The Boomerang Attack. In: Knudsen, L.R. (ed.) FSE 1999. LNCS, vol. 1636, pp. 156–170. Springer, Heidelberg (1999)
5. Biryukov, A., Nikolić, I., Roy, A.: Boomerang Attacks on BLAKE-32. In: Joux, A. (ed.) FSE 2011. LNCS, vol. 6733, pp. 218–237. Springer, Heidelberg (2011)
6. Lamberger, M., Mendel, F.: Higher-Order Differential Attack on Reduced SHA-256. Cryptology ePrint Archive: Report 2011/037, http://eprint.iacr.org/
7. Biryukov, A., Lamberger, M., Mendel, F., Nikolić, I.: Second-Order Differential Collisions for Reduced SHA-256. In: Lee, D.H. (ed.) ASIACRYPT 2011. LNCS, vol. 7073, pp. 270–287. Springer, Heidelberg (2011)
8. Mendel, F., Nad, T.: Boomerang Distinguisher for the SIMD-512 Compression Function. In: Bernstein, D.J., Chatterjee, S. (eds.) INDOCRYPT 2011. LNCS, vol. 7107, pp. 255–269. Springer, Heidelberg (2011)
9. Sasaki, Y.: Boomerang Distinguishers on MD4-Family: First Practical Results on Full 5-Pass HAVAL. In: Miri, A., Vaudenay, S. (eds.) SAC 2011. LNCS, vol. 7118, pp. 1–18. Springer, Heidelberg (2012)
10. Sasaki, Y., Wang, L.: 2-Dimension Sums: Distinguishers Beyond Three Rounds of RIPEMD-128 and RIPEMD-160, http://eprint.iacr.org/2012/049.pdf
11. Aumasson, J.-P., Çalık, Ç., Meier, W., Özen, O., Phan, R.C.-W., Varıcı, K.: Improved Cryptanalysis of Skein. In: Matsui, M. (ed.) ASIACRYPT 2009. LNCS, vol. 5912, pp. 542–559. Springer, Heidelberg (2009)
12. Leurent, G., Roy, A.: Boomerang Attacks on Hash Function Using Auxiliary Differentials. In: Dunkelman, O. (ed.) CT-RSA 2012. LNCS, vol. 7178, pp. 215–230. Springer, Heidelberg (2012)
13. Chen, J., Jia, K.: Improved Related-Key Boomerang Attacks on Round-Reduced Threefish-512. In: Kwak, J., Deng, R.H., Won, Y., Wang, G. (eds.) ISPEC 2010. LNCS, vol. 6047, pp. 1–18. Springer, Heidelberg (2010)
14. Yu, H., Chen, J., Wang, X.: The Boomerang Attacks on the Round-Reduced Skein-512. In: Knudsen, L.R., Wu, H. (eds.) SAC 2012. LNCS, vol. 7707, pp. 288–304. Springer, Heidelberg (2012)
15. Khovratovich, D., Nikolić, I.: Rotational Cryptanalysis of ARX. In: Hong, S., Iwata, T. (eds.) FSE 2010. LNCS, vol. 6147, pp. 333–346. Springer, Heidelberg (2010)
16. Biryukov, A., Wagner, D.: Slide Attacks. In: Knudsen, L.R. (ed.) FSE 1999. LNCS, vol. 1636, pp. 245–259. Springer, Heidelberg (1999)
17. Yoshida, H., Biryukov, A.: Analysis of a SHA-256 Variant. In: Preneel, B., Tavares, S. (eds.) SAC 2005. LNCS, vol. 3897, pp. 245–260. Springer, Heidelberg (2006)
18. 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)
19. Wagner, D.: A Generalized Birthday Problem. In: Yung, M. (ed.) CRYPTO 2002. LNCS, vol. 2442, pp. 288–303. Springer, Heidelberg (2002)

Provable Security of BLAKE with Non-ideal Compression Function

Elena Andreeva, Atul Luykx, and Bart Mennink

Dept. Electrical Engineering, ESAT/COSIC, KU Leuven, and IBBT, Belgium
{elena.andreeva,atul.luykx,bart.mennink}@esat.kuleuven.be

Abstract. We analyze the security of the SHA-3 finalist BLAKE. The BLAKE hash function follows the HAIFA design methodology, and as such it achieves optimal preimage, second preimage and collision resistance, and is indifferentiable from a random oracle up to approximately $2^{n/2}$ assuming the underlying compression function is ideal.

In our work we show, however, that the compression function employed by BLAKE exhibits a non-random behavior and is in fact differentiable in only $2^{n/4}$ queries. Our attack undermines the provable security strength of BLAKE in the ideal compression function model, not only with respect to its overall indifferentiability but also its collision and (second) preimage security. Our next contribution is the restoration of the security results for BLAKE in the ideal model by refining the level of modularity and assuming that BLAKE's underlying block cipher is an ideal cipher. We prove that BLAKE is optimally collision, second preimage, and preimage secure (up to a constant). We go on to show that BLAKE is still indifferentiable from a random oracle up to the old bound of $2^{n/2}$ queries, albeit under a weaker assumption: the ideality of its block cipher.

1 Introduction

Hash functions are a main building block for numerous cryptographic applications. Due to a series of attacks on the widely deployed SHA-1 hash function by Wang et al. [20], the US National Institute for Standards and Technology (NIST) recommended the replacement of SHA-1 by the SHA-2 hash function family and announced a call for the design of a new SHA-3 hashing algorithm in 2007 [17]. Five candidates, BLAKE [4], Grøstl [13], JH [21], Keccak [7] and Skein [12], made it to the third and final round of the competition. Evaluating the security of the remaining five SHA-3 candidates is crucial and particularly relevant in the ongoing process for the selection of the finalist hash function due by the end of 2012.

The focus of this work is the provable security of the BLAKE SHA-3 hash function candidate. To assess the security of BLAKE, we follow the security criteria listed by NIST in their call for a new SHA-3 hash function: collision, second preimage, preimage security and resistance to the length extension attacks. The BLAKE hash function is designed by Aumasson et al. [4], and follows the HAIFA design methodology of Biham and Dunkelman [8]. Its underlying compression function f employs internally a block cipher E and exhibits some similarities with the Davies-Meyer compression function. As described in the SHA-3 provable security survey by Andreeva et al. [2], BLAKE inherits preimage, second preimage, collision, and indifferentiability security guarantees of the HAIFA design, assuming ideality of the underlying compression function. More precisely, due to the specific HAIFA counter BLAKE is indifferentiable from a random oracle in the indifferentiability framework of Maurer et al. [15]. The advantage of an adversary against the collision and (second) preimage security of BLAKE is upper bounded by approximately $q^2/2^n$ and $q/2^n$, respectively. While all of these results are true in the ideal compression function model, no concrete (in)differentiability results are known for the BLAKE compression function, which is the main motivation for this work.

Our Contributions. Firstly, in Sect. 3 we present an attack on the BLAKE compression function f, which shows that f is differentiable from a random oracle in $2^{n/4}$ queries. This is less than ideally expected, and as a consequence the existing BLAKE indifferentiability bound, together with the collision and (second) preimage security bounds from [2], are reduced by a square root. These findings yield a provable security level that is not compliant with the NIST security requirements. The indifferentiability attack is a serious motivation for carrying out further security analysis of the BLAKE mode of operation in a way that restores its security guarantees. One approach in this direction is to refine the level of modularity in the security analysis and to investigate security properties of the BLAKE mode of operation under the assumption that the underlying block cipher E, rather than the compression function, is ideal.

This brings us to our second contribution, which is presented in Sect. 4. In the ideal cipher model, we conclude optimal (up to a constant) collision and (everywhere) preimage security of f. This result is important to establish a strong confidence in the security of f in the sense that even if f exhibits some non-ideal behavior, its collision and preimage security are not compromised when E behaves close to ideal. Furthermore, due to the collision and everywhere preimage resistance preservation of the HAIFA design [3], the BLAKE mode of operation inherits the optimal security of f with respect to both properties.

Next, in Sect. 5 we reconsider the second preimage resistance of BLAKE. As a HAIFA design, BLAKE does not preserve second preimage resistance [3], and proving second preimage security of BLAKE's compression function does not directly translate to the second preimage security of BLAKE. To assess the second preimage security of BLAKE, we therefore analyze directly the BLAKE mode of operation in the ideal cipher model and prove it optimally (everywhere) second preimage resistant, up to a constant. This result confirms BLAKE's resistance against the second preimage attacks of Dean [11] and Kelsey and Schneier [14], even when the non-ideal compression function of BLAKE is employed.

Finally, in Sect. 6 we restore the indifferentiability result of BLAKE to the old bound of approximately $2^{n/2}$ queries by giving a proof with an ideal underlying block cipher E. We show that despite the differentiability of BLAKE's compression function f, in the ideal cipher model the BLAKE mode of operation does not suffer structural design flaws. We summarize our results on BLAKE in Table 1.

Our results amount to an important contribution to the security analysis of the SHA-3 finalist BLAKE in a way that addresses all the security criteria of NIST listed in their call for a new SHA-3 hash function. We provide a thorough investigation of these security properties for BLAKE in the ideal block cipher model.

Table 1. A summary of our results on the BLAKE hash function $\mathcal{H}$ and its compression function f in the ideal block cipher model. The bounds denote the required number of queries to mount an attack.

	preimage	second preimage	collision	indifferentiability
f	$\Theta(2^n)$ Sect. 4	–	$\Theta(2^{n/2})$ Sect. 4	$O(2^{n/4})$ Sect. 3
$\mathcal{H}$	$\Theta(2^n)$ Sect. 4	$\Theta(2^n)$ Sect. 5	$\Theta(2^{n/2})$ Sect. 4	$\Omega(2^{n/2})$ Sect. 6

2 Preliminaries

For $n \in \mathbb{N}$, let $\{0,1\}^n$ denote the set of bit strings of length n and let $\{0,1\}^*$ denote the set of bit strings of arbitrary length. For two bit strings x, y, $x\|y$ denotes their concatenation and $x \oplus y$ their bitwise XOR. By $[x]_2 = x\|x$ we denote the concatenation of two copies of x. If x is of even length, then x^l and x^r denote its left and right halves where $|x^l| = |x^r|$. For natural m, n, $\langle m \rangle_n$ is the encoding of m as an n-bit string. We denote by $\text{Bloc}(2n)$ the set of all block ciphers $E : \{0,1\}^{2n} \times \{0,1\}^{2n} \to \{0,1\}^{2n}$, where the first input corresponds to the key input. A random oracle [5] is a function which provides a random output for each new query. A random $2n$-bit block cipher is a block cipher randomly sampled from $\text{Bloc}(2n)$. A random primitive will also be called "ideal".

2.1 BLAKE

In accordance with the SHA-3 hash function specification, BLAKE supports outputs of size $n = 224$, 256, 384, and 512 bits. In this work we focus on the variants $n = 256, 512$, as the 224- and 384-variants are chopped versions of these.

BLAKE takes as input a salt s of $n/2$ bits (chosen by the user), and a message M of arbitrary length. The evaluation of $\mathcal{H}(s, M)$ is done as follows. Firstly, the message M is padded into message blocks $m_1, \ldots, m_k$ of $2n$ bits, where the padding function is defined as $\mathsf{pad}(M) = M\|10^{-|M|-n/2-2 \bmod 2n}1\|\langle |M| \rangle_{n/2}$. Along with these message blocks, counter blocks $t_1, \ldots, t_k$ of length $n/4$ bits are generated. This counter keeps track of the number of message bits hashed so far and equals 0 if the i-th message block contains no message bits[1]. Starting

[1] In more detail, $t_i = \langle i2n \rangle_{n/4}$ if $i2n \le |M|$, $t_i = \langle |M| \rangle_{n/4}$ if $(i-1)2n < |M| \le i2n$, and $t_i = \langle 0 \rangle_{n/4}$ if $|M| \le (i-1)2n$.

from an initial state value $h_0 \in \{0,1\}^n$, the message blocks m_i and counter blocks t_i are compressed iteratively into the state using a compression function $f : \{0,1\}^n \times \{0,1\}^{n/2} \times \{0,1\}^{2n} \times \{0,1\}^{n/4} \to \{0,1\}^n$. Here, the second input to f denotes the salt s. The output of the BLAKE hash function is defined as its final state value $\mathcal{H}(s, M) = h_k$.

The compression function f uses a block cipher $E : \{0,1\}^{2n} \times \{0,1\}^{2n} \to \{0,1\}^{2n}$ and a constant $C \in \{0,1\}^n$. A diagram of the function is displayed in Fig. 1 and an algorithmic description follows.

```
procedure $f(h_{i-1}, s, m_i, t_i)$
    $v_i \leftarrow (h_{i-1} \| s \| [t_i^l]_2 \| [t_i^r]_2) \oplus (0^n \| C)$
    $w_i \leftarrow E(m_i, v_i)$
    $h_i \leftarrow w_i^l \oplus w_i^r \oplus h_{i-1} \oplus [s]_2$
    return $h_i$
end procedure
```

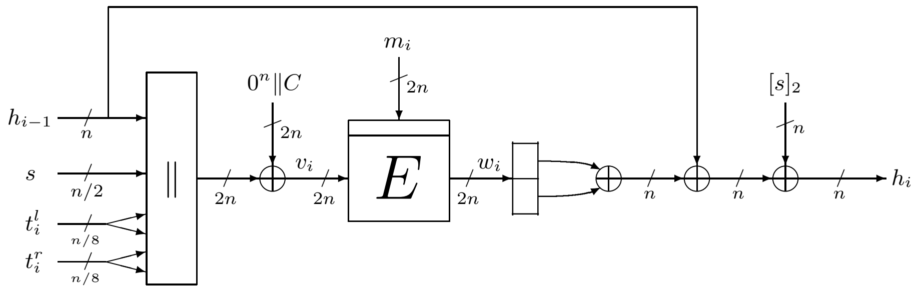


Fig. 1. The BLAKE compression function f of Sect. 2.1

2.2 Preimage, Second Preimage and Collision Security

An adversary $\mathcal{A}$ is a probabilistic algorithm with oracle access to a randomly sampled block cipher $E \xleftarrow{\$} \text{Bloc}(2n)$. In this work, we consider information-theoretic adversaries only. This type of adversary has unbounded computational power, and its complexity is measured by the number of queries made to its oracle. The adversary can make queries to E and its inverse E^{-1}. These queries are stored in a query history $\mathcal{Q}$ as elements of the form (m_j, v_j, w_j), where j is the query index, m_j is the key input to the block cipher (note that for BLAKE the message input to f is the key input to E), and v_j and w_j denote the plaintext and ciphertext, respectively. Associated to query (m_j, v_j, w_j) we define the value $x_j = w_j^l \oplus w_j^r \oplus h_j \oplus [s_j]_2$ as the output of the compression function f, where we parse $h_j \| s_j \| t_j^{(1)} \| t_j^{(2)} \| t_j^{(3)} \| t_j^{(4)} \leftarrow v_j \oplus (0^n \| C)$. In the remainder, we assume that $\mathcal{Q}$ always contains the queries required for the attack and that the adversary never makes queries to which it knows the answer in advance.

Let $F : \{0,1\}^p \to \{0,1\}^n$ for $p \geq n$ be a compressing function instantiated with a randomly chosen block cipher $E \xleftarrow{\$} \text{Bloc}(2n)$. In this work, F will either be the BLAKE mode of operation $\mathcal{H}$ or its compression function f. For the preimage and second preimage security analysis in this work, we consider the notion of everywhere preimage and second preimage resistance [19]. In the ideal model setting (where randomness is gained from the ideal primitive rather than from the use of an explicit random key), these notions are the strongest options because they guarantee preimage (resp. second preimage) security for every range (resp. domain) point.

Definition 1. *The advantage of an everywhere preimage finding adversary $\mathcal{A}$ is defined as*

$$\mathbf{Adv}_F^{\mathrm{epre}}(\mathcal{A}) = \max_{y \in \{0,1\}^n} \mathbf{Pr}\left(E \overset{\$}{\leftarrow} \mathrm{Bloc}(2n),\ z \leftarrow \mathcal{A}^{E,E^{-1}}(y)\ :\ F(z) = y \right).$$

We define by $\mathbf{Adv}_F^{\mathrm{epre}}(q)$ the maximum advantage of any adversary making q queries to its oracles.

Definition 2. *Let $\lambda \leq p$. The advantage of an everywhere second preimage finding adversary $\mathcal{A}$ is defined as*

$$\mathbf{Adv}_F^{\mathrm{esec}[\lambda]}(\mathcal{A}) = \max_{z' \in \{0,1\}^\lambda} \mathbf{Pr}\left(\begin{array}{r} E \overset{\$}{\leftarrow} \mathrm{Bloc}(2n),\ z \leftarrow \mathcal{A}^{E,E^{-1}}(z')\ : \\ z \neq z' \ \wedge\ F(z) = F(z') \end{array} \right).$$

We define by $\mathbf{Adv}_F^{\mathrm{esec}[\lambda]}(q)$ the maximum advantage of any adversary making q queries to its oracles.

In case F denotes the BLAKE compression function f of Sect. 2.1, its domain points are of the form $z = (h, s, m, t)$. If F is the BLAKE mode of operation $\mathcal{H}$, its domain points are parsed as $z = (s, M) \in \{0,1\}^{n/2} \times \{0,1\}^*$, where in the second preimage notion λ is required to be of length at least $n/2$ bits.

We define the collision security of a compressing function F as follows.

Definition 3. *Fix a constant $h_0 \in \{0,1\}^n$. The advantage of a collision finding adversary $\mathcal{A}$ is defined as*

$$\mathbf{Adv}_F^{\mathrm{col}}(\mathcal{A}) = \mathbf{Pr}\left(\begin{array}{r} E \overset{\$}{\leftarrow} \mathrm{Bloc}(2n),\ z, z' \leftarrow \mathcal{A}^{E,E^{-1}}\ : \\ z \neq z' \ \wedge\ F(z) \in \{F(z'), h_0\} \end{array} \right).$$

We define by $\mathbf{Adv}_F^{\mathrm{col}}(q)$ the maximum advantage of any adversary making q queries to its oracles.

As before, in case F denotes the compression function f the strings z and z' are of the form (h, s, m, t) and (h', s', m', t'), and if F is the BLAKE mode of operation, z and z' are parsed as (s, M) and (s', M'). Note that in all definitions $\mathcal{A}$ can freely choose the salt, e.g. a collision with different salts is counted valid. Our results directly apply to the setting of a fixed salt.

2.3 Indifferentiability

The indifferentiability framework introduced by Maurer et al. [15] is an extension of the classical notion of indistinguishability. It proves that if a construction $\mathcal{C}^{\mathcal{G}}$ based on an ideal subcomponent $\mathcal{G}$ is indifferentiable from an ideal primitive $\mathcal{R}$, then $\mathcal{C}^{\mathcal{G}}$ can replace $\mathcal{R}$ in any system. Although recent results by Ristenpart et al. [18] show that indifferentiability does not capture all properties of a random oracle, indifferentiability still remains the best way to rule out structural attacks for a large class of hash function applications.

Definition 4. *A Turing machine $\mathcal{C}$ with oracle access to an ideal primitive $\mathcal{G}$ is called $(t_D, t_S, q, \varepsilon)$ indifferentiable from an ideal primitive $\mathcal{R}$ if there exists a simulator $\mathcal{S}$, such that for any distinguisher $\mathcal{D}$ we have*

$$\mathbf{Adv}_{\mathcal{C}}^{\mathrm{pro}}(\mathcal{D}) = \left|\mathbf{Pr}\left(\mathcal{D}^{\mathcal{C}^{\mathcal{G}},\mathcal{G}} = 1\right) - \mathbf{Pr}\left(\mathcal{D}^{\mathcal{R},\mathcal{S}^{\mathcal{R}}} = 1\right)\right| < \varepsilon.$$

The simulator has oracle access to $\mathcal{R}$ and runs in time at most t_S. The distinguisher runs in time at most t_D and makes at most q queries.

In the remainder, we refer to $\mathcal{C}^{\mathcal{G}}, \mathcal{G}$ as the "real world", and to $\mathcal{R}, \mathcal{S}^{\mathcal{R}}$ as the "simulated world"; the distinguisher $\mathcal{D}$ converses either with the real or the simulated world and its goal is to tell both worlds apart. $\mathcal{D}$ can query both its "left oracle" L (either $\mathcal{C}$ or $\mathcal{R}$) and its "right oracle" R (either $\mathcal{G}$ or $\mathcal{S}$).

For the purpose of the presented indifferentiability results, $\mathcal{G}$ throughout denotes a random block cipher $E \xleftarrow{\$} \mathrm{Bloc}(2n)$. $\mathcal{C}$ is either the BLAKE mode of operation $\mathcal{H}$ or its compression function f, and $\mathcal{R}$ will be a random oracle RO with the same domain and range as $\mathcal{C}$.

3 Differentiability of *f*

We consider the indifferentiability of the BLAKE compression function f from a random oracle RO (with the same domain and range as f), when the underlying block cipher E is sampled uniformly at random $E \xleftarrow{\$} \mathrm{Bloc}(2n)$. In more detail, we construct a distinguisher $\mathcal{D}$, such that for any simulator $\mathcal{S}$, $\mathcal{D}$ differentiates (f, E) from $(RO, \mathcal{S})$ in about $2^{n/4}$ queries, hence significantly faster than expected.

The differentiability attack considers fixed-points for f, by which we in this case mean values (h, m, s, t) such that $f(h, m, s, t) = h \oplus [s]_2$. The presence of fixed-points of these form have already been pointed out in [4, Sect. 5.2.4]. However, not every block cipher evaluation renders a valid compression function evaluation, due to the duplication of t in the block cipher input. The simulator may be able to take advantage of this, which makes the attack proof more complicated.

Theorem 1. *Let $E \xleftarrow{\$} \mathrm{Bloc}(2n)$, and let $RO : \{0,1\}^{n+n/2+2n+n/4} \to \{0,1\}^n$ be a random compression function. For any simulator $\mathcal{S}$ that makes at most*

$q_{\mathcal{S}} \leq 2^{n-3}$ queries to RO, there exists a distinguisher $\mathcal{D}$ that makes at most $2^{n/4}+1$ queries to its oracles, such that

$$\mathbf{Adv}_f^{\mathrm{pro}}(\mathcal{D}) \geq 1 - e^{-1} - \frac{q_{\mathcal{S}}}{2^n} \geq 0.5.$$

Proof. Let $\mathcal{S}$ be any simulator making at most $q_{\mathcal{S}}$ queries to RO. We construct a distinguisher $\mathcal{D}$ that differentiates (f, E) from $(RO, \mathcal{S})$ with a significant probability. $\mathcal{D}$ has query access to (L, R, R^{-1}) (either (f, E, E^{-1}) or $(RO, \mathcal{S}, \mathcal{S}^{-1})$) and operates as follows.

1. $\mathcal{D}$ picks $2^{n/4}$ distinct messages m_j, queries $v_j \leftarrow R^{-1}(m_j, 0)$, and parses $h_j \| s_j \| t_j^{(1)} \| t_j^{(2)} \| t_j^{(3)} \| t_j^{(4)} \leftarrow v_j \oplus (0^n \| C)$;
2. If $t_j^{(1)} \| t_j^{(3)} \neq t_j^{(2)} \| t_j^{(4)}$ for all $j \in \{1, \ldots, 2^{n/4}\}$, $\mathcal{D}$ guesses $(L, R) = (RO, \mathcal{S})$ and halts;
3. Let $j \in \{1, \ldots, 2^{n/4}\}$ be such that $t_j^{(1)} \| t_j^{(3)} = t_j^{(2)} \| t_j^{(4)}$. $\mathcal{D}$ queries $h \leftarrow L(h_j, s_j, m_j, t_j^{(1)} \| t_j^{(3)})$, and guesses $(L, R) = (f, E)$ if and only if $h = h_j \oplus [s_j]_2$.

$\mathcal{D}$ guesses its oracles correctly *except* if one of the following events occur:

$$\begin{array}{lll} \mathsf{E}_1: & \forall\, j: \ t_j^{(1)} \| t_j^{(3)} \neq t_j^{(2)} \| t_j^{(4)} & \big|\ (L, R) = (f, E); \\ \mathsf{E}_2: & \exists\, j: \ t_j^{(1)} \| t_j^{(3)} = t_j^{(2)} \| t_j^{(4)} \text{ and } h = h_j \oplus [s_j]_2 & \big|\ (L, R) = (RO, \mathcal{S}). \end{array}$$

In particular, $\mathbf{Adv}_f^{\mathrm{pro}}(\mathcal{D}) \geq 1 - \mathbf{Pr}(\mathsf{E}_1) - \mathbf{Pr}(\mathsf{E}_2)$. We start with $\mathbf{Pr}(\mathsf{E}_2)$, and we suppose $(L, R) = (RO, \mathcal{S})$. E_2 in fact covers the event that $\mathcal{S}$ finds a fixed-point for RO, namely inputs h_j, s_j, m_j, t_j such that $RO(h_j, s_j, m_j, t_j) = h_j \oplus [s_j]_2$. As $\mathcal{S}$ makes at most $q_{\mathcal{S}}$ queries, it can find such fixed-point with probability at most $q_{\mathcal{S}}/2^n$. Next, we consider $\mathbf{Pr}(\mathsf{E}_1)$, and we suppose $(L, R) = (f, E)$. As E is a random block cipher and the message blocks m_j are all different, the probabilities $\mathbf{Pr}\left(t_j^{(1)} \| t_j^{(3)} \neq t_j^{(2)} \| t_j^{(4)}\right)$ are independent for different indices j, and satisfy

$$\mathbf{Pr}\left(t_j^{(1)} \| t_j^{(3)} \neq t_j^{(2)} \| t_j^{(4)}\right) = 1 - \mathbf{Pr}\left(t_j^{(1)} \| t_j^{(3)} = t_j^{(2)} \| t_j^{(4)}\right) = 1 - 1/2^{n/4}.$$

Therefore, $\mathbf{Pr}(\mathsf{E}_1) = \left(1 - 1/2^{n/4}\right)^{2^{n/4}} \leq e^{-1}$. We thus obtain $\mathbf{Adv}_f^{\mathrm{pro}}(\mathcal{D}) \geq 1 - e^{-1} - q_{\mathcal{S}}/2^n \geq 0.5$ for $q_{\mathcal{S}} \leq 2^{n-3}$. $\square$

4 Collision and Preimage Resistance of f and $\mathcal{H}$

In this section, we analyze the collision and (everywhere) preimage resistance of the BLAKE compression function f. We achieve optimal security (up to a constant). As the HAIFA mode of operation preserves collision and everywhere preimage resistance [3], these results directly carry over to the BLAKE hash function $\mathcal{H}$. The proofs of Thms. 2 and 3 can be found in the full version of this paper [1].

Theorem 2. *Let $n \in \mathbb{N}$. The advantage of any adversary $\mathcal{A}$ in finding a collision for f after $q < 2^{2n-1}$ queries can be upper bounded by $\mathbf{Adv}_f^{\mathrm{col}}(q) \leq \frac{q(q+1)}{2^n}$.*

Theorem 3. *Let $n \in \mathbb{N}$. The advantage of any adversary $\mathcal{A}$ in finding a preimage for f after $q < 2^{2n-1}$ queries can be upper bounded by $\mathbf{Adv}_f^{\mathrm{epre}}(q) \leq \frac{2q}{2^n}$.*

5 Second Preimage Resistance of $\mathcal{H}$

Due to the lack of second preimage security preservation [3] of the BLAKE hash function $\mathcal{H}$, we investigate the second preimage security of $\mathcal{H}$ directly, rather than its compression function (as in the collision and preimage cases). Our proof shows similarities with the second preimage proof for HAIFA by Bouillaguet and Fouque [9]. Our proof, however, is realized in the ideal cipher (rather than ideal compression function) model.

Theorem 4. *Let $n \in \mathbb{N}$ and $\lambda \geq n/2$. The advantage of any adversary $\mathcal{A}$ in finding a second preimage for $\mathcal{H}$ after $q < 2^{2n-1}$ queries can be upper bounded by*

$$\mathbf{Adv}_{\mathcal{H}}^{\mathrm{esec}[\lambda]}(q) \leq \frac{4q}{2^n}.$$

Proof. Let $(s', M') \in \{0,1\}^{n/2} \times \{0,1\}^{\lambda - n/2}$ be the target preimage. We denote $\mathsf{pad}(M') = m'_1 \| \cdots \| m'_{k'}$ and denote by $t'_1, \ldots, t'_{k'}$ the corresponding counter values. Note that by construction, $t'_{i'} = \langle i'2n \rangle_{n/4}$ for $i' \in \{1, \ldots, k'-2\}$, $t'_{k'} \neq \langle k'2n \rangle_{n/4}$, and $t'_{k'-1}$ may or may not be of the form $\langle (k'-1)2n \rangle_{n/4}$. The block cipher executions corresponding to this hash function evaluation are given to the adversary for free. That is, $\mathcal{A}$ is forced to make the k' corresponding queries but will not be charged for this. Denote by $h'_0, \ldots, h'_{k'}$ the state values corresponding to the evaluation of $\mathcal{H}(s', M')$.

The goal of the adversary is to find a tuple $(s, M) \neq (s', M')$ such that $\mathcal{H}(s, M) = \mathcal{H}(s', M')$ and such that the query history contains all block cipher evaluations required for the computation of $\mathcal{H}(s, M)$. We pose the following claim.

Claim. Suppose $\mathcal{A}$ finds $(s, M) \neq (s', M')$ such that $\mathcal{H}(s, M) = \mathcal{H}(s', M')$. Denote by $m_1, \ldots, m_k$, $t_1, \ldots, t_k$, and $h_0, \ldots, h_k$ the message blocks, counter values and intermediate state values corresponding to the computation of $\mathcal{H}(s, M)$. There must be $i \in \{1, \ldots, k\}$ and $i' \in \{1, \ldots, k'\}$ such that $f(h_{i-1}, s, m_i, t_i) = f(h'_{i'-1}, s', m'_{i'}, t'_{i'})$, where $(h_{i-1}, s, m_i) \neq (h'_{i'-1}, s', m'_{i'})$ and $t_i, t'_{i'}$ satisfy

$$t_i = t'_{i'} \text{ or } \Big(t_i \neq \langle i2n \rangle_{n/4} \text{ and } t'_{i'} \neq \langle i'2n \rangle_{n/4} \Big). \tag{1}$$

Proof (Proof of claim). As $\mathcal{H}(s, M) = \mathcal{H}(s', M')$, we have $h_k = h'_{k'}$. If $|M| \neq |M'|$, then $m_k \neq m'_{k'}$, $t_k \neq \langle k2n \rangle_{n/4}$ and $t'_{k'} \neq \langle k'2n \rangle_{n/4}$, and a collision of the prescribed form is found. Thus, suppose $|M| = |M'|$. This implies $k = k'$ and $t_i = t'_i$ for $i = 1, \ldots, k$. If $s \neq s'$, a collision for h_k is directly found. If $s = s'$, we necessarily have $M \neq M'$ and by the standard collision resistance preservation proof for the Merkle-Damgård mode of operation (see e.g. [2,3,10,16]), there must by an index i such that $f(h_{i-1}, s, m_i, t_i) = f(h'_{i-1}, s', m'_i, t'_i)$ but $(h_{i-1}, m_i) \neq (h'_{i-1}, m'_i)$. This completes the proof. $\square$

It consequently suffices to consider the probability of the adversary finding a collision with any of the k' compression function evaluations of $\mathcal{H}(s', M')$, such that the corresponding counter values satisfy (1). We call a collision of this form a "valid collision". Thus,

$$\mathbf{Adv}_{\mathcal{H}}^{\mathrm{esec}[\lambda]}(q) \leq \sum_{j=1}^{q} \mathbf{Pr}\left(j\text{-th query is valid collision}\right). \tag{2}$$

Let $j = 1, \ldots, q$. We consider the probability that the j-th query results in a collision with any of the target state values. We distinguish between *forward* and *inverse* queries.

Valid collision by a forward query. The adversary makes an encryption query of the form (m_j, v_j) to receive a ciphertext w_j such that $E(m_j, v_j) = w_j$. Parse $h_j \| s_j \| t_j^{(1)} \| t_j^{(2)} \| t_j^{(3)} \| t_j^{(4)} \leftarrow v_j \oplus (0^n \| C)$. If $t_j^{(1)} \| t_j^{(3)} \neq t_j^{(2)} \| t_j^{(4)}$ the query does not correspond to a compression function evaluation and a valid collision is obtained with probability 0. Hence, we assume $t_j^{(1)} \| t_j^{(3)} = t_j^{(2)} \| t_j^{(4)}$. In this case, the query corresponds to the compression function evaluation $f(h, s, m, t_j^{(1)} \| t_j^{(3)}) = w_j^l \oplus w_j^r \oplus h_j \oplus [s_j]_2$.

If $t_j^{(1)} \| t_j^{(3)} = \langle \alpha 2n \rangle_{n/4}$ for some $\alpha \in \{1, \ldots, k'-1\}$, this means the query corresponds to a compression function at the α-th position, and (1) may be satisfied only for $i' = \alpha$. If $t_j^{(1)} \| t_j^{(3)} \neq \alpha 2n$ for any $\alpha \in \{1, \ldots, k'-2\}$, (1) can be satisfied only for $i' \in \{k'-1, k'\}$. In any other case, there is no i' that makes (1) satisfied. In any case, there are at most 2 values that $w_j^l \oplus w_j^r \oplus h_j \oplus [s_j]_2$ may hit in order to render a valid collision. As in the proof of Thm. 2, the j-th query results in a valid collision with probability at most $\frac{2 \cdot 2^n}{2^{2n-q}}$.
Valid collision by a inverse query. The analysis follows the same lines. The j-th query results in a valid collision with probability at most $\frac{2 \cdot 2^{3n/4}}{2^{2n}-q}$.

A valid collision for the compression function f is generated by either a forward or inverse query, and the j-th query thus renders a valid collision with probability at most $\max\left\{\frac{2 \cdot 2^n}{2^{2n}-q}, \frac{2 \cdot 2^{3n/4}}{2^{2n}-q}\right\} = \frac{2 \cdot 2^n}{2^{2n}-q}$. Summing over all q queries, we obtain from (2):

$$\mathbf{Adv}_{\mathcal{H}}^{\mathrm{esec}[\lambda]}(q) \leq \sum_{j=1}^{q} \frac{2 \cdot 2^n}{2^{2n}-q} \leq \frac{2q2^n}{2^{2n}-q} \leq \frac{4q}{2^n},$$

where the last inequality holds as $q < 2^{2n-1}$. $\square$

6 Indifferentiability of $\mathcal{H}$

We show that the BLAKE mode of operation is indifferentiable from a random oracle in the ideal cipher model. To this end, we construct a simulator such that any distinguisher requires at least approximately $2^{n/2}$ queries to differentiate $(\mathcal{H}, E, E^{-1})$ from $(RO, \mathcal{S}, \mathcal{S}^{-1})$.

Theorem 5. *Let $E \xleftarrow{\$} \mathrm{Bloc}(2n)$, and let RO be a random oracle. Let $\mathcal{D}$ be any distinguisher that makes at most q_L left queries of maximal length $2n \cdot \ell$ bits (not including the salt), q_R queries to R and R^{-1}, and runs in time t. Then*

$$\mathbf{Adv}^{\mathrm{pro}}_{\mathcal{H}}(\mathcal{D}) \leq 5\frac{q(q+1)}{2^n},$$

where $q = \ell q_L + q_R$ and $\mathcal{S}$ is the simulator of Fig. 2, which makes less than q_R queries to RO and runs in time $O(q_R^2)$.

The remainder of this section is devoted to the proof of Thm. 5. In Sect. 6.1, we introduce some additional definitions required for the proof. The simulator used in the proof is introduced and formalized in Sect. 6.2. Then, Thm. 5 is formally proven in Sect. 7.

6.1 Definitions

To facilitate the analysis, we rewrite the BLAKE padding function pad', such that on input of a tuple $(s, M) \in \{0,1\}^{n/2} \times \{0,1\}^*$ it is defined as

$$\mathsf{pad}'(s, M) = (s\|m_1\|t_1) \| \cdots \| (s\|m_k\|t_k),$$

with $m_1\| \cdots \|m_k = \mathsf{pad}(M)$ and the t_i calculated appropriately based on the m_i. The strings $s\|m_i\|t_i$ are called augmented message blocks which we will denote by a_i. We analyze the BLAKE hash function with pad' padding and respectively its compression function f that accepts inputs of the form (h, a), with $h \in \{0,1\}^n$, and $a \in \{0,1\}^{n/2+2n+n/4}$ the augmented message.

Let $V := \{0,1\}^{2n}$ be the plain- and ciphertext space of both $\mathcal{S}(m, \cdot)$ and $E(m, \cdot)$. We define $\beta_{h,s} : V \to \{0,1\}^n$ as

$$\beta_{h,s}(w) = w^l \oplus w^r \oplus h \oplus [s]_2.$$

For $h' \in \{0,1\}^n$, we define $\beta_{h,s}^{-1}(h') = \{w \in \{0,1\}^{2n} \mid w^l \oplus w^r \oplus h \oplus [s]_2 = h'\}$. We call these sets the *fibers* of $\beta_{h,s}$. Note that each fiber is of size 2^n.

For the construction of the simulator, we will maintain an initially empty table T, in which all query-response tuples (m, v, w) of $\mathcal{S}$ and $\mathcal{S}^{-1}$ are stored. We write $T_m^+(v) = w$ and $T_m^-(w) = v$. Associated to T we define a graph G, which is initialized with the single node h_0. The compression function evaluations corresponding to the entries of T are maintained in G. The domain and range of $\mathcal{S}$ and $\mathcal{S}^{-1}$ are not intermediate state values and a query-response might not necessarily correspond to a path in G. In fact, a query-response tuple (m, v, w)

corresponds to a compression function evaluation and can be converted into a path in G if and only if v can be parsed as $h\|s\|[t^l]_2\|[t^r]_2 \leftarrow v \oplus (0^n\|C)$. In this case, the tuple corresponds to the following path in G:

$$h \xrightarrow{s\|m\|t} \beta_{h,s}(w).$$

Note that if G contains a path $h_0 \xrightarrow{a_1} h_1 \cdots \xrightarrow{a_k} h_k$, this implies T contains all queries required for the evaluation of $f(\ldots f(f(h_0, a_1), a_2) \ldots, a_k) = h_k$, when f is instantiated with $\mathcal{S}$.

6.2 Simulator

Designing the simulator comes down to making sure that $(RO, \mathcal{S}, \mathcal{S}^{-1})$ matches $(\mathcal{H}, E, E^{-1})$ as closely as possible. Notice that for $(\mathcal{H}, E, E^{-1})$ an $\mathcal{H}$ query is a chain of E queries which can be converted to

$$h_0 \xrightarrow{a_1} h_1 \cdots \xrightarrow{a_k} h_k,$$

with $a_1\| \cdots \|a_k = \mathsf{pad}'(s, M)$ for some $s \in \{0,1\}^{n/2}$ and $M \in \{0,1\}^*$; it is this property that the simulator should mimic. What this essentially means is that the simulator needs to carefully handle queries that may extend the set of nodes reachable from h_0. For any other query, it suffices for the simulator to respond randomly.

For simplicity and to improve the readability of the simulator, we opt for a simulator that behaves like a random function. That is, when generating a random answer it will be sampled from V, therewith allowing collisions in T. Clearly, this will result in a higher success probability for the distinguisher, but because the elements of V are of size $2n$ bits (while the hash function has range $\{0,1\}^n$), this security loss will be negligible. This loss will be reflected in the bound obtained in Sect. 7.

Recall from Sect. 6.1 that a query-response tuple (m, v, w) adds an edge to the graph if and only if v can be parsed as $h\|s\|[t^l]_2\|[t^r]_2 \leftarrow v \oplus (0^n\|C)$ and for now we simply assume this to be true, adding the edge $h \xrightarrow{s\|m\|t} h' = \beta_{h,s}(w)$ to G. The main purpose of $\mathcal{S}$ is to maintain consistency for the paths leaving from h_0. Thus, we investigate how the simulator handles queries extending any of these paths.

In case of inverse queries, note that h depends on the output of the simulator, v. If the simulator generates v uniformly at random, the edge extends a path from h_0 only if h hits any node already reachable from h_0. This case occurs with small probability, and we can safely have the simulator respond randomly on an inverse query. The resulting security loss is reflected in the obtained indifferentiability bound derived in Sect. 7.

In case of forward queries, h and $a = s\|m\|t$ are determined by the inputs by the distinguisher, and thus it may force the number of nodes reachable from h_0 to increase. Suppose a path $h_0 \xrightarrow{a_1} h_1 \cdots \xrightarrow{a_k} h_k = h$ is in G. We distinguish among the following cases:

- $a_1\|\cdots\|a_k\|a = \mathsf{pad}'(s,M)$ for some $s \in \{0,1\}^{n/2}$ and $M \in \{0,1\}^*$. The simulator should assure consistency with RO, hence its answer w should comply with $\beta_{h,s}(w) = RO(s,M)$;
- $a_1\|\cdots\|a_k\|a \neq \mathsf{pad}'(s,M)$ for any $s \in \{0,1\}^{n/2}$ and $M \in \{0,1\}^*$. There is no consistency required, and the simulator responds randomly.

Two peculiarities may occur in case of a forward query of this form. At first, it may be the case that a newly added edge extends to two different paths. This would however mean a compression function collision has occurred, an event that happens with small probability and results in a security loss in the final indifferentiability bound obtained in Sect. 7. Secondly, the value $h' = \beta_{h,s}(w)$ may hit a node in the graph, in which case the path will be increased with two edges. A similar reasoning as for inverse queries applies here: h' hits another node in the graph with small probability only, and the simulator does not need to handle this situation.

The formal description of the simulator is given in Fig. 2. It uses the following procedure.

```
procedure FINDPATHS($h, a$)
    $P \leftarrow \emptyset$                    ▷ will contain paths and corresponding messages
    for all paths $h_0 \xrightarrow{a_1} h_1 \cdots \xrightarrow{a_k} h_k$ in $G$ do
        if $h = h_k$ and $\exists\, M$ such that $\mathsf{pad}'(s,M) = a_1\|\cdots\|a_k\|a$ then
            $P \leftarrow P \cup \left\{\left(M, h_0 \xrightarrow{a_1} h_1 \cdots \xrightarrow{a_k} h_k\right)\right\}$
        end if
    end for
    return $P$
end procedure
```

Lemma 1. *If the simulator does not return a response retrieved from the table T, the response will be uniformly distributed over V.*

Proof. With a forward query there are two cases, one where the simulator queries RO and one where it does not. If the simulator does not query RO, then by definition it responds uniformly over V. If the simulator does query RO, then it receives an $h' = RO(s,M)$ uniformly distributed over $\{0,1\}^n$. As the fibers of $\beta_{h,s}$ form a partition of V, uniformly selecting an element from an arbitrary fiber of $\beta_{h,s}$ is the same as uniformly selecting an element from V.

Since the inverse queries of the simulator are by definition uniformly distributed over V, we attain our result. □

Now, the formal proof of Thm. 5 is given in Sect. 7: by a game hopping argument we prove that $(RO, \mathcal{S}^{RO}, (\mathcal{S}^{RO})^{-1})$ is indistinguishable from $(\mathcal{H}^E, E, E^{-1})$, where $\mathcal{S}$ is the simulator introduced in this section.

7 Proof of Thm. 5

In this section, we will bound the advantage of any distinguisher in differentiating the simulated world (with the simulator of Fig. 2) from the real world. The

Simulator Forward Query

```
1: procedure $\mathcal{S}(m, v)$
2:     if $T_m^+(v) = \perp$ then
3:         $h \| s \| t^{(1)} \| t^{(2)} \| t^{(3)} \| t^{(4)} \leftarrow v \oplus (0^n \| C)$
4:         $w \xleftarrow{\$} V$
5:         if $t^{(1)} \| t^{(3)} = t^{(2)} \| t^{(4)}$ then
6:             $P \leftarrow \textsc{findPaths}(h, s \| m \| t^{(1)} \| t^{(3)})$
7:             if $P \neq \emptyset$ then
8:                 $(M, path) \xleftarrow{\$} P$
9:                 $w \xleftarrow{\$} \beta_{h,s}^{-1}(RO(s, M))$
10:            end if
11:            add $h \xrightarrow{s \| m \| t^{(1)} \| t^{(3)}} \beta_{h,s}(w)$ to $G$
12:        end if
13:        $T_m^+(v) \leftarrow w$
14:    end if
15:    return $T_m^+(v)$
16: end procedure
```

Simulator Inverse Query

```
1: procedure $\mathcal{S}^{-1}(m, w)$
2:     if $T_m^-(w) = \perp$ then
3:         $v \leftarrow T_m^-(w) \xleftarrow{\$} V$
4:         $h \| s \| t^{(1)} \| t^{(2)} \| t^{(3)} \| t^{(4)} \leftarrow v \oplus (0^n \| C)$
5:         if $t^{(1)} \| t^{(3)} = t^{(2)} \| t^{(4)}$ then
6:             add $h \xrightarrow{s \| m \| t^{(1)} \| t^{(3)}} \beta_{h,s}(w)$ to $G$
7:         end if
8:     end if
9:     return $T_m^-(w)$
10: end procedure
```

Random Oracle

```
1: procedure $RO(s, M)$
2:     if $F[s, M] = \perp$ then          ▷ $F$ is an array
3:         $F[s, M] \xleftarrow{\$} \{0, 1\}^n$
4:     end if
5:     return $F[s, M]$
6: end procedure
```

Fig. 2. The definition of the simulator $\mathcal{S}$ used in the proof of Thm. 5, and the random oracle RO

proof of indifferentiability consists of a sequence of six games where we specify three algorithms (L_i, R_i, R_i^{-1}) (for $i = 1, \ldots, 6$) with which a distinguisher can interact. These games are given in Fig. 3. The first game corresponds to the simulated world and the sixth game corresponds to the real world $(\mathcal{H}^E, E, E^{-1})$. By G_i we denote the event $\mathcal{D}^{L_i, R_i, R_i^{-1}} = 1$. Clearly,

$$\mathbf{Adv}_{\mathcal{H}}^{\mathrm{pro}}(\mathcal{D}) = |\mathbf{Pr}(G_1) - \mathbf{Pr}(G_6)| \leq \sum_{i=1}^{5} |\mathbf{Pr}(G_i) - \mathbf{Pr}(G_{i+1})| . \qquad (3)$$

In the remainder of this section, the distances between the adjacent games will be bounded, and the claim of Thm. 5 will be directly obtained from (3).

Games 1 and 2

The first game is $(RO, \mathcal{S}^{RO}, (\mathcal{S}^{RO})^{-1})$. The biggest change in the second game is that $\mathcal{H}^{\mathcal{S}}$ is called in L_2 in line 2. Note that the result of $\mathcal{H}^{\mathcal{S}}$ is not used and L_2 returns a value generated by RO, i.e. the responses of L_1 and L_2 are identical. Yet, calling $\mathcal{H}^{\mathcal{S}}$ still has side effects on game 2 as the simulator's table and graph are updated based on the $\mathcal{S}$ queries made by $\mathcal{H}^{\mathcal{S}}$. As a result the simulator in game 2 gains more knowledge than the simulator in game 1. In particular, we will *mark* each query-response from all calls of $\mathcal{S}$ in $\mathcal{H}^{\mathcal{S}}$ whenever a query has not been made before by $\mathcal{D}$; these marked query-responses in T represent the extra knowledge gained by the simulator in game 2. Queries made by $\mathcal{D}$ are made unmarkable in line 12 in R_2 and line 22 in R_2^{-1}, these queries provide the simulator with no extra information compared to the simulator of game 1.

Game 1

```
1: procedure $L_1(s, M)$
2:     return $RO(s, M)$
3: end procedure

4: procedure $R_1(m, v)$
5:     return $\mathcal{S}(m, v)$
6: end procedure

7: procedure $R_1^{-1}(m, w)$
8:     return $\mathcal{S}^{-1}(m, w)$
9: end procedure
```

Game 2

```
1: procedure $L_2(s, M)$
2:     mark all $(m, v, w)$ used in $\mathcal{H}^{\mathcal{S}}(s, M)$
3:     return $RO(s, M)$
4: end procedure

5: procedure $R_2(m, v)$
6:     if $(m, v, T_m^+(v))$ is marked then
7:         delete $(m, v, T_m^+(v))$ from $T$
8:         delete corresponding path from $G$
9:     end if
10:    DELETEMARKEDPATHS$(m, v)$
11:    $w \leftarrow S(m, v)$
12:    make $(m, v, w)$ unmarkable
13:    return $w$
14: end procedure

15: procedure $R_2^{-1}(m, w)$
16:    if $(m, T_m^-(w), w)$ is marked then
17:        bad $\leftarrow$ true
18:        delete $(m, T_m^-(w), w)$ from $T$
19:        delete corresponding path from $G$
20:    end if
21:    $v \leftarrow S^{-1}(m, w)$
22:    make $(m, v, w)$ unmarkable
23:    return $v$
24: end procedure
```

Game 3

```
1: procedure $L_3(s, M)$
2:     return $L_2(s, M)$
3: end procedure

4: procedure $R_3(m, v)$
5:     return $R_2(m, v)$
6: end procedure

7: procedure $R_3^{-1}(m, w)$
8:     if $(m, T_m^-(w), w)$ is marked then
9:         bad $\leftarrow$ true
10:    end if
11:    return $\mathcal{S}^{-1}(m, w)$
12: end procedure
```

Game 4

```
1: procedure $L_4(s, M)$
2:     $\mathcal{H}^{R_4}(s, M)$
3:     return $RO(s, M)$
4: end procedure

5: procedure $R_4(m, v)$
6:     $w_{\text{temp}} \leftarrow T_m^+(v)$
7:     $w \leftarrow S(m, v)$
8:     if $w_{\text{temp}} = \perp$ then
9:         bad $\leftarrow$ ISCOLLISION$(m, v, T_m^+(v))$
10:    end if
11:    return $w$
12: end procedure

13: procedure $R_4^{-1}(m, w)$
14:    return $\mathcal{S}^{-1}(m, w)$
15: end procedure
```

Game 5

```
1: procedure $L_5(s, M)$
2:     return $\mathcal{H}^{R_5}(s, M)$
3: end procedure

4: procedure $R_5(m, v)$
5:     return $R_4(m, v)$
6: end procedure

7: procedure $R_5^{-1}(m, w)$
8:     return $R_4^{-1}(m, v)$
9: end procedure
```

Game 6

```
1: procedure $L_6(s, M)$
2:     return $\mathcal{H}^{R_6}(s, M)$
3: end procedure

4: procedure $R_6(m, v)$
5:     return $E(m, v)$
6: end procedure

7: procedure $R_6^{-1}(m, w)$
8:     return $E^{-1}(m, w)$
9: end procedure
```

Fig. 3. Games $1, \ldots, 6$ used in the proof of Thm. 5

Note that if we ignore the lines of code in R_2 and R_2^{-1} dealing with marked query-responses we are left with lines 11, 13, 21, and 23, where we see that R_2 and R_2^{-1} simply query $\mathcal{S}$ and $\mathcal{S}^{-1}$ and return the result. The rest of the code is there in order to undo the side effects of $\mathcal{H}^{\mathcal{S}}$. A first step in removing the side effects of $\mathcal{H}^{\mathcal{S}}$ is, when a marked query is made by $\mathcal{D}$, to remove the knowledge $\mathcal{S}$ has of that query (implemented in the if-statements) and then to re-query $\mathcal{S}$ again. This does not deal with all possible cases because we know that sometimes $\mathcal{S}$ queries RO in order to ensure consistency. In particular, in game 1 the distinguisher could know that a particular intermediate value should map to the result of some L_1 response while the simulator does not know this, resulting in a collision. Yet, this collision could be avoided if the simulator knows all of the intermediate values used through a call to $\mathcal{H}^{\mathcal{S}}$. To this end, R_2 employs the following procedure so that $\mathcal{S}$ "forgets" the intermediate values:

```
**procedure** deleteMarkedPaths($m, v$)
    $h\|s\|t^{(1)}\|t^{(2)}\|t^{(3)}\|t^{(4)} \leftarrow v \oplus (0^n\|C)$
    **if** $t^{(1)}\|t^{(3)} \neq t^{(2)}\|t^{(4)}$ **then**
        **return**
    **end if**
    $P \leftarrow$ findPaths($h, s\|m\|t^{(1)}\|t^{(3)}$)
    **for all** $(M, h_0 \xrightarrow{a_1} \cdots \xrightarrow{a_k} h_k) \in P$ **do**
        **for** $i \leftarrow 0, \ldots, k-1$ **do**
            **if** $(m, v, w)$ associated with $h_i \xrightarrow{a_{i+1}} h_{i+1}$ is marked **then**
                delete $(m, v, w)$ from $T$
                delete $h_i \xrightarrow{a_{i+1}} h_{i+1}$ from $G$
            **end if**
        **end for**
    **end for**
**end procedure**
```

When invoked through a forward query, deleteMarkedPaths checks for all paths to which this particular query extends using the findPaths procedure and deletes any marked query-responses used along these paths, thereby eliminating all marked intermediate values used by a $\mathcal{H}^{\mathcal{S}}$ query.

Now we take a look at R_2^{-1} and see exactly how the $\mathcal{H}^{\mathcal{S}}$ query is dealt with. If the query-response $(m, T_m^-(w), w)$ is not marked, then either (m, w) has never been queried before or the distinguisher has queried (m, w) before; in either case we get the exact same behavior as R_1^{-1}. If $(m, T_m^-(w), w)$ is marked, then this means that the distinguisher has not queried (m, w) and that $\mathcal{H}^{\mathcal{S}}$ has queried (m, w). Removing $(m, T_m^-(w), w)$ from T and G and then querying $\mathcal{S}^{-1}(m, w)$ will return some uniformly chosen response from V. This is the same as never having queried $\mathcal{S}^{-1}(m, w)$ and then querying it, meaning we get the same behavior out of $\mathcal{S}^{-1}$ in R_2^{-1} as in R_1^{-1}. Note that the newly generated $\mathcal{S}^{-1}(m, w)$ very likely differs from the value previously generated (when it was queried by $\mathcal{H}^{\mathcal{S}}$). However, as L_2 never discloses the data from $\mathcal{H}^{\mathcal{S}}$, this is not a problem.

Finally we just need to compare R_1 with R_2. Say that $(m, v, T_m^+(v))$ is unmarked, i.e. $\mathcal{H}^{\mathcal{S}}$ has never queried (m, v). When calling deleteMarkedPaths, there are a few possibilities:

- findPaths returns the empty set. The subsequent call to $\mathcal{S}$ will return some arbitrary element of V, as would exactly happen in R_1;
- findPaths finds some valid path, but there are no marked query-responses along this path. This means that the distinguisher has queried the full path itself and $\mathcal{S}$ will respond similarly in both R_1 and R_2;
- findPaths finds a valid path and there are marked query-responses along this path, but these are removed. Thus, the simulator call from R_2 has the same amount of information as the simulator call from R_1.

If $(m, v, T_m^+(v))$ is marked, then knowledge of that particular query-response is removed. In effect we are then dealing with an unmarked query-response $(m, v, \perp)$ and are reduced to the case above.

We have shown that each R_2 and R_2^{-1} query will execute the same code within $\mathcal{S}$ as each R_1 and R_1^{-1} query, respectively, and since they all return the response of the $\mathcal{S}$ query, we have $\mathbf{Pr}(G_1) = \mathbf{Pr}(G_2)$.

Games 2 and 3

Note that L_2 and L_3, and R_2 and R_3 are exactly the same, so we need to compare the responses of R_2^{-1} and R_3^{-1}. It is clear that R_2^{-1} and R_3^{-1} are identical until **bad**. The **bad** event corresponds to $\mathcal{H}^{\mathcal{S}}$ first querying $\mathcal{S}$ resulting in the query-response (m, v, w) with path $h_1 \xrightarrow{a} h_2$, the distinguisher guessing this particular w correctly from the set $\beta_{h_1,s}^{-1}(h_2)$, and finally $\mathcal{D}$ calling $R_i^{-1}(m, w)$ without explicitly calling $R_i(m, v)$ (otherwise $R_i(m, v)$ would unmark (m, v, w)). Since every fiber of $\beta_{h_1,s}^{-1}$ has size 2^n, an upper bound for the probability of finding such a w is $q_R/2^n$, as q_R bounds the number of right oracle inverse queries by $\mathcal{D}$. Therefore, as **bad** can be triggered in both games, $|\mathbf{Pr}(G_2) - \mathbf{Pr}(G_3)| \leq 2\frac{q_R}{2^n}$.

Games 3 and 4

The following procedure is used in game 4 to detect collisions:

```
procedure isCollision($m, v, w$)
    $h \| s \| t^{(1)} \| t^{(2)} \| t^{(3)} \| t^{(4)} \leftarrow v$
    if $t^{(1)} \| t^{(3)} \neq t^{(2)} \| t^{(4)}$ then
        return false
    end if
    $h' \leftarrow \beta_{h,s}(w)$
    return (number of edges connected to $h'$) $> 1$        ▷ returns true/false
end procedure
```

Since R_4 is identical to $\mathcal{S}$, L_3 and L_4 are identical. The only difference between games 3 and 4 can be found in R_3 and R_4, yet this is the same difference as between R_1 and R_2 and we may conclude that $\mathbf{Pr}(G_3) = \mathbf{Pr}(G_4)$.

Games 4 and 5

The difference between games 4 and 5 lies in the response given by the left oracles: game 4 uses RO while game 5 uses $\mathcal{H}^{R_5}$. We will show that as long as **bad** is not triggered, the responses of both left oracles are the same.

Lemma 2. *As long as* **bad** *is not set to true,* $L_4(s,M) = L_5(s,M)$.

Proof. We can write $\mathcal{H}^{R_i}(s,M)$, with i equal to 4 or 5, as $h_0 \xrightarrow{a_1} h_1 \cdots \xrightarrow{a_k} h_k$, with $a_1\|\cdots\|a_k = \mathsf{pad}'(s,M)$. If none of the nodes h_j for $j > 0$ are in G, then $\mathcal{H}^{R_i}$ will query R_i in sequence starting from h_0 and ending up at $RO(s,M)$ since the simulator will learn the entire message M in sequence by the time $h_{k-1} \xrightarrow{a_k} h_k$ is queried and can respond with $RO(s,M)$.

On the other hand, if there is some node h_j in G, then it must be the case that the particular path $h_{j-1} \xrightarrow{a_j} h_j$ is in G, otherwise $\mathcal{H}^{R_i}(s,M)$ will trigger **bad**. Furthermore $h_{j-2} \xrightarrow{a_{j-1}} h_{j-1}$ must have been queried before the $h_{j-1} \xrightarrow{a_j} h_j$ query:

– if $h_{j-2} \xrightarrow{a_{j-1}} h_{j-1}$ is not in G then $\mathcal{H}^{R_i}(s,M)$ will place it in G resulting in a collision because h_{j-1} is already in G, and
– if $h_{j-2} \xrightarrow{a_{j-1}} h_{j-1}$ is in G then it must have occurred before the a_j query since the result of the a_{j-1} query would otherwise have ended up as a node in G.

This means that R_i receives each of the $h_{j-1} \xrightarrow{a_j} h_j$ queries in order from $j = 1$ to k and can respond consistently with $RO(s,M)$. □

By the collision resistance of the BLAKE compression function (Sect. 4), the probability of a collision occurring is upper bounded by $2q(q+1)/2^n$. Hence, as **bad** can be triggered in both games, $|\mathbf{Pr}(G_4) - \mathbf{Pr}(G_5)| \leq 4\dfrac{q(q+1)}{2^n}$.

Games 5 and 6

The right oracles of game 6 form a permutation for each message, whereas the right oracles of game 5 do not. By Lem. 1, the right oracles of game 5 are uniformally distributed over V (R_5 and R_5^{-1} are essentially just the simulator), which means that the difference between game 5 and game 6 is the difference between a permutation and a random function, which we know is bounded as follows [6]: $|\mathbf{Pr}(G_5) - \mathbf{Pr}(G_6)| \leq \dfrac{q_R(q_R-1)}{2^{2n}}$.

Acknowledgments. This work has been funded in part by the IAP Program P6/26 BCRYPT of the Belgian State (Belgian Science Policy), in part by the European Commission through the ICT program under contract ICT-2007-216676 ECRYPT II, and in part by the Research Council K.U.Leuven: GOA TENSE. The third author is supported by a Ph.D. Fellowship from the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen).

References

1. Andreeva, E., Luykx, A., Mennink, B.: Provable security of BLAKE with non-ideal compression function. Cryptology ePrint Archive, Report 2011/620 (2011); Full version of this paper

2. Andreeva, E., Mennink, B., Preneel, B.: Security Reductions of the Second Round SHA-3 Candidates. In: Burmester, M., Tsudik, G., Magliveras, S., Ilić, I. (eds.) ISC 2010. LNCS, vol. 6531, pp. 39–53. Springer, Heidelberg (2011)
3. Andreeva, E., Neven, G., Preneel, B., Shrimpton, T.: Seven-Property-Preserving Iterated Hashing: ROX. In: Kurosawa, K. (ed.) ASIACRYPT 2007. LNCS, vol. 4833, pp. 130–146. Springer, Heidelberg (2007)
4. Aumasson, J., Henzen, L., Meier, W., Phan, R.: SHA-3 proposal BLAKE (2010); Submission to NIST's SHA-3 competition
5. Bellare, M., Rogaway, P.: Random oracles are practical: A paradigm for designing efficient protocols. In: ACM Conference on Computer and Communications Security, pp. 62–73. ACM Press, New York (1993)
6. Bellare, M., Rogaway, P.: Code-based game-playing proofs and the security of triple encryption. Cryptology ePrint Archive, Report 2004/331 (2004)
7. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: The KECCAK sponge function family (2011); Submission to NIST's SHA-3 competition
8. Biham, E., Dunkelman, O.: A framework for iterative hash functions – HAIFA. Cryptology ePrint Archive, Report 2007/278 (2007)
9. Bouillaguet, C., Fouque, P.: Practical hash functions constructions resistant to generic second preimage attacks beyond the birthday bound (2010); Submitted to Information Processing Letters
10. Damgård, I.B.: A Design Principle for Hash Functions. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 416–427. Springer, Heidelberg (1990)
11. Dean, R.: Formal Aspects of Mobile Code Security. PhD thesis, Princeton University, Princeton (1999)
12. Ferguson, N., Lucks, S., Schneier, B., Whiting, D., Bellare, M., Kohno, T., Callas, J., Walker, J.: The Skein Hash Function Family (2010); Submission to NIST's SHA-3 competition
13. Gauravaram, P., Knudsen, L., Matusiewicz, K., Mendel, F., Rechberger, C., Schläffer, M., Thomsen, S.: Grøstl – a SHA-3 candidate (2011); Submission to NIST's SHA-3 competition
14. 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)
15. Maurer, U.M., Renner, R.S., 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)
16. Merkle, R.C.: One Way Hash Functions and DES. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 428–446. Springer, Heidelberg (1990)
17. National Institute for Standards and Technology: Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA3) family (2007)
18. Ristenpart, T., Shacham, H., Shrimpton, T.: Careful with Composition: Limitations of the Indifferentiability Framework. In: Paterson, K.G. (ed.) EUROCRYPT 2011. LNCS, vol. 6632, pp. 487–506. Springer, Heidelberg (2011)
19. 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)
20. 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)
21. Wu, H.: The Hash Function JH (2011); Submission to NIST's SHA-3 competition

TWINE: A Lightweight Block Cipher for Multiple Platforms

Tomoyasu Suzaki, Kazuhiko Minematsu, Sumio Morioka, and Eita Kobayashi

NEC Corporation, 1753 Shimonumabe, Nakahara-Ku, Kawasaki, Japan
{t-suzaki,k-minematsu,s-morioka,e-kobayashi}@jp.nec.com

Abstract. This paper presents a 64-bit lightweight block cipher TWINE supporting 80 and 128-bit keys. TWINE realizes quite small hardware implementation similar to the previous lightweight block cipher proposals, yet enables efficient software implementations on various CPUs, from micro-controllers to high-end CPUs. This characteristic is obtained by the use of generalized Feistel combined with an improved block shuffle, introduced at FSE 2010.

Keywords: lightweight block cipher, generalized Feistel, block shuffle.

1 Introduction

Motivation. Recent advances in tiny computing devices, such as RFID and sensor network nodes, give rise to the need of symmetric encryption with highly-limited resources, called lightweight encryption. While AES has been widely deployed, it is often inappropriate for such small devices due to their size/power/memory constraints, even though there are constant efforts for small-footprint AES, e.g., [13,30,39]. To fill the gap, many hardware-oriented lightweight block ciphers have been recently proposed, e.g., [8,12,17,18,20,22,23,26,40,44], and more.

In this paper, we propose TWINE, a new lightweight 64-bit block cipher. Our primary goal is to achieve hardware efficiency equivalent to previous proposals, and at the same time good software performance on various CPUs, from low-end micro-controllers to high-end ones (such as Intel Core-i series). For this purpose, we avoid the hardware-oriented design options, most notably a bit permutation, and build a block cipher using 4-bit components.

Design. Specifically, we employ Type-2 generalized Feistel structure [45], GFS for short, with 16 nibble-blocks. The drawback of such design is a poor diffusion property, resulting in a small-but-slow cipher due to quite many rounds. To overcome the problem, we employ the idea of Suzaki and Minematsu at FSE '10 [42] which substantially improves diffusion by using a different block shuffle from the original cyclic shift. As a result, TWINE is also efficient on software and enables compact unification of encryption and decryption. The features of TWINE are (1) no bit permutation, (2) generalized Feistel-based, and (3) no Galois-Field matrix. The components are only one 4-bit S-box, XOR, and 4-bit-wise permutation (shuffle). As far as we know, this is the first attempt that unifies these three features. There is a predecessor called LBlock [44] which has some resemblances to ours, however TWINE is an independent work and has several concrete design advantages (See Section 3).

Implementation. We implemented TWINE on hardware and software. Our hardware implementations suggest that the encryption-only TWINE can be implemented with $1,503$ Gate Equivalent (GE), and a serialized implementation results in $1,011$ GEs using a shared sbox architecture. For both cases, we did not consider the hard-wired key or special key signaling (as employed by [40]). These figures are comparable to, or even better than, the leading hardware-oriented proposals, in particular when a standard key treatment is required.

On 8-bit micro-controllers, TWINE is implemented within 0.8 to 1.5 Kbytes ROM. The speed is relatively fast compared to other lightweight ciphers. We also tried implementations on 32 and 64-bit CPUs. Due to the nature of GFS (and the use of identical 4-bit S-box), TWINE is quite easy to implement using a SIMD instruction doing a vector-permutation, which we call vector-permutation instruction (VPI). Starting from Hamburg's works on AES [19], VPI has been recognized as a powerful tool for fast cryptography (e.g. [1, 10, 11]), and we find that VPI extremely works fine with TWINE. For example, on Intel Core-i5 U560 we observed 4.75 cycles/byte[1] using VPI called `pshufb`. This figure is quite impressive in the realm of (lightweight) block ciphers. For reference, we observed that AES using VPI [19] runs at 6.66 cycles/byte on the same processor. As our VPI-based implementation has a quite simple structure, it is easy to understand and port to other CPUs. TWINE's well-balanced performance under multiple platforms makes it suitable to heterogeneous networks, consisting of (e.g.) a huge number of tiny sensor nodes which independetly encrypt sensor information and one server computer which performs the information aggregation and decryption.

Security. As TWINE is a variant of GFS it is definitely important to evaluate the security against attacks suitable to GFS, such as the impossible differential cryptanalysis (IDC) and the saturation cryptanalysis (SC). We perform a thorough analysis (as a new cipher proposal) on TWINE including IDC and SC, and present IDC against 23-round TWINE-80 and 24-round TWINE-128 as the most powerful attacks we have found so far. The attack is fully exploits the key schedule, and can be seen as an interesting example of highly-optimized IDC against GFS-based ciphers.

The organization of the paper is as follows. In Section 2 we describe the specification of TWINE. Section 3 explains the design rationale for TWINE. Section 4 presents the results of security evaluation, and Section 5 presents the implementation results of both hardware and software. Section 6 concludes the paper.

2 Specification of TWINE

Notations. A bitwise exclusive-OR is denoted by $\oplus$. For binary strings, x and y, $x \| y$ denotes their concatenation. Let $|x|$ denote the bit length of x. If $|x| = m$,

[1] In a double-block encryption. See Section 5.2.

we may write $x_{(m)}$ to emphasize its bit length. If $|x| = 4c$ for a positive integer c, we write $x \to (x_0\|x_1\|\ldots\|x_{c-1})$, where $|x_i| = 4$, is the partition operation into the 4-bit sub-blocks. The opposite operation, $(x_0\|x_1\|\ldots\|x_{c-1}) \to x$, is similarly defined. The partition operation may be implicit, i.e., we may simply write x_i to denote the i-th 4-bit subsequence for any $4c$-bit string x.

Data Processing Part. TWINE is a 64-bit block cipher with 80 or 128-bit key. We write TWINE-80 or TWINE-128 to denote the key length. The global structure of TWINE is a variant of Type-2 GFS [41,45] with 16 4-bit sub-blocks. A round function of TWINE consists of a nonlinear layer using 4-bit S-boxes and a diffusion layer, which permutes the 16 blocks. Unlike original Type-2 GFS, the diffusion layer is not a cyclic shift and is chosen to provide a better diffusion than the cyclic shift from the result of [42]. This round function is iterated for 36 times for both key lengths, where the diffusion layer of the last round is omitted. For $i = 1, \ldots, 36$, i-th round uses a 32-bit round key, RK^i, which is derived from the secret key, $K_{(n)}$ with $n \in \{80, 128\}$, using the key schedule. The encryption process is written as Algorithm 2.1.

The data processing part essentially consists of a 4-bit S-box, denoted by S, and a permutation of block indexes, $\pi : \{0, \ldots, 15\} \to \{0, \ldots, 15\}$, where j-th sub-block is mapped to $\pi[j]$-th sub-block. The figure of the round function is in Fig. 1. The decryption of TWINE uses the same S-box and key schedule as used in the encryption, with the inverse block shuffle. See Algorithm 2.2.

Key Schedule Part. The key schedule produces $RK_{(32\times 36)}$ from the secret key, $K_{(n)}$, for $n \in \{80, 128\}$. It is a variant of GFS with few S-boxes (the same as one used at the data processing). The 80-bit key schedule uses 6-bit round constants, $\mathrm{CON}^i_{(6)} = \mathrm{CON}^i_{H(3)}\|\mathrm{CON}^i_{L(3)}$ for $i = 1$ to 35, and $Rot\mathbf{z}(x)$ means z-bit left cyclic shift of x. Its pseudocode is in Algorithm 2.3. For 128-bit key, see Appendix A. We remark that CON^i corresponds to 2^i in $\mathrm{GF}(2^6)$ with primitive polynomial $z^6 + z + 1$.

Algorithm 2.1: TWINE.ENC$(P_{(64)}, RK_{(32\times 36)}, C_{(64)})$

```
$X^1_{0(4)}\|X^1_{1(4)}\|\ldots\|X^1_{14(4)}\|X^1_{15(4)} \leftarrow P$,   $\mathrm{RK}^1_{(32)}\|\ldots\|\mathrm{RK}^{36}_{(32)} \leftarrow RK_{(32\times 36)}$
for $i \leftarrow 1$ to 35
  do { $\mathrm{RK}^i_{0(4)}\|\mathrm{RK}^i_{1(4)}\|\ldots\|\mathrm{RK}^i_{6(4)}\|\mathrm{RK}^i_{7(4)} \leftarrow \mathrm{RK}^i_{(32)}$
       for $j \leftarrow 0$ to 7 do  $X^i_{2j+1} \leftarrow S(X^i_{2j} \oplus \mathrm{RK}^i_j) \oplus X^i_{2j+1}$
       for $h \leftarrow 0$ to 15 do  $X^{i+1}_{\pi[h]} \leftarrow X^i_h$
for $j \leftarrow 0$ to 7 do  $X^{36}_{2j+1} \leftarrow S(X^{36}_{2j} \oplus \mathrm{RK}^{36}_j) \oplus X^{36}_{2j+1}$
$C \leftarrow X^{36}_0\|X^{36}_1\|\ldots\|X^{36}_{14}\|X^{36}_{15}$
```

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S(x)$	C	0	F	A	2	B	9	5	8	3	D	7	1	E	6	4

h	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi[h]$	5	0	1	4	7	12	3	8	13	6	9	2	15	10	11	14

```
**Algorithm 2.2:** TWINE.Dec($C_{(64)}, RK_{(32\times 36)}, P_{(64)}$)

$X^{36}_{0(4)} \| X^{36}_{1(4)} \| \ldots \| X^{36}_{14(4)} \| X^{36}_{15(4)} \leftarrow C$,   $\mathrm{RK}^{1}_{(32)} \| \ldots \| \mathrm{RK}^{36}_{(32)} \leftarrow RK_{(32\times 36)}$
for $i \leftarrow 36$ to $2$
    do
        $\mathrm{RK}^{i}_{0(4)} \| \mathrm{RK}^{i}_{1(4)} \| \ldots \| \mathrm{RK}^{i}_{6(4)} \| \mathrm{RK}^{i}_{7(4)} \leftarrow \mathrm{RK}^{i}_{(32)}$
        for $j \leftarrow 0$ to $7$ do   $X^{i}_{2j+1} \leftarrow S(X^{i}_{2j} \oplus \mathrm{RK}^{i}_{j}) \oplus X^{i}_{2j+1}$
        for $h \leftarrow 0$ to $15$ do   $X^{i-1}_{\pi^{-1}[h]} \leftarrow X^{i}_{h}$
for $j \leftarrow 0$ to $7$ do   $X^{1}_{2j+1} \leftarrow S(X^{1}_{2j} \oplus \mathrm{RK}^{1}_{j}) \oplus X^{1}_{2j+1}$
$P \leftarrow X^{1}_{0} \| X^{1}_{1} \| \ldots \| X^{1}_{14} \| X^{1}_{15}$
```

h	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi^{-1}[h]$	1	2	11	6	3	0	9	4	7	10	13	14	5	8	15	12

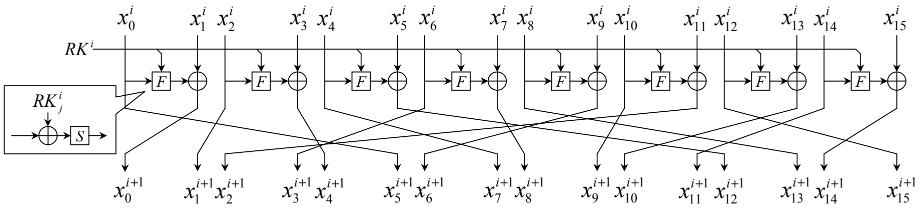

x_0^i x_1^i x_2^i x_3^i x_4^i x_5^i x_6^i x_7^i x_8^i x_9^i x_{10}^i x_{11}^i x_{12}^i x_{13}^i x_{14}^i x_{15}^i
RK^i
F
RK_j^i
S
x_0^{i+1} x_1^{i+1} x_2^{i+1} x_3^{i+1} x_4^{i+1} x_5^{i+1} x_6^{i+1} x_7^{i+1} x_8^{i+1} x_9^{i+1} x_{10}^{i+1} x_{11}^{i+1} x_{12}^{i+1} x_{13}^{i+1} x_{14}^{i+1} x_{15}^{i+1}

Fig. 1. Round function of TWINE

```
**Algorithm 2.3:** TWINE.KeySchedule-80($K_{(80)}, RK_{(32\times 36)}$)

$\mathrm{WK}_{0(4)} \| \mathrm{WK}_{1(4)} \| \ldots \| \mathrm{WK}_{18(4)} \| \mathrm{WK}_{19(4)} \leftarrow K$
for $r \leftarrow 1$ to $35$
    do
        $\mathrm{RK}^{r}_{(32)} \leftarrow \mathrm{WK}_1 \| \mathrm{WK}_3 \| \mathrm{WK}_4 \| \mathrm{WK}_6 \| \mathrm{WK}_{13} \| \mathrm{WK}_{14} \| \mathrm{WK}_{15} \| \mathrm{WK}_{16}$
        $\mathrm{WK}_1 \leftarrow \mathrm{WK}_1 \oplus S(\mathrm{WK}_0)$,   $\mathrm{WK}_4 \leftarrow \mathrm{WK}_4 \oplus S(\mathrm{WK}_{16})$
        $\mathrm{WK}_7 \leftarrow \mathrm{WK}_7 \oplus 0 \| CON^{r}_{H}$,   $\mathrm{WK}_{19} \leftarrow \mathrm{WK}_{19} \oplus 0 \| CON^{r}_{L}$
        $\mathrm{WK}_0 \| \cdots \| \mathrm{WK}_3 \leftarrow Rot4(\mathrm{WK}_0 \| \cdots \| \mathrm{WK}_3)$
        $\mathrm{WK}_0 \| \cdots \| \mathrm{WK}_{19} \leftarrow Rot16(\mathrm{WK}_0 \| \cdots \| \mathrm{WK}_{19})$
$\mathrm{RK}^{36}_{(32)} \leftarrow \mathrm{WK}_1 \| \mathrm{WK}_3 \| \mathrm{WK}_4 \| \mathrm{WK}_6 \| \mathrm{WK}_{13} \| \mathrm{WK}_{14} \| \mathrm{WK}_{15} \| \mathrm{WK}_{16}$
$RK \leftarrow \mathrm{RK}^1 \| \mathrm{RK}^2 \| \ldots \| \mathrm{RK}^{35} \| \mathrm{RK}^{36}$
```

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
CON^i	01	02	04	08	10	20	03	06	0C	18	30	23	05	0A	14	28	13	26
i	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	
CON^i	0F	1E	3C	3B	35	29	11	22	07	0E	1C	38	33	25	09	12	24	

3 Design Rationale

3.1 Basic Objective

Our goal is to build a lightweight block cipher enabling compact hardware comparable to previous proposals, while keeping the efficiency on multiple CPUs, from low-end microcontroller to general-purpose 32/64-bit CPU.

On LBlock. We remark that LBlock [44], proposed independently of ours, is quite similar to our proposal. It is a 64-bit block cipher using a variant of balanced Feistel whose round function consists of 8 4-bit S-boxes and a nibble-wise permutation and a 8-bit cyclic shift. Such a structure can be transformed into a structure proposed at [42], though we do not know whether the authors of [44] are aware of it. We investigated LBlock in this respect and found that the LBlock's diffusion layer is equivalent to that of the decryption of TWINE. Note that this choice is reasonable from Table 6 of [42], as it satisfies both of the fastest diffusion and the highest immunities against linear and differential attacks among other block shuffles.

Nevertheless, there are important differences between TWINE and LBlock[2]. First, LBlock uses ten distinct S-boxes while TWINE uses single S-box. TWINE's design contributes to a compact serialized hardware and fast software (indeed, our fast SIMD implementation was impossible if multiple S-boxes were used). Second, LBlock uses a bit permutation in its key scheduling, which decreases software efficiency.

3.2 Parameters and Components

Rounds. As far as we investigated, the most powerful attack against TWINE is a dedicated impossible differential attack, which breaks 23-round TWINE-80 and 24-round TWINE-128. From this, we consider 36-round TWINE-128 has a sufficient security margin. Employing the same 36-round for TWINE-80 may look slight odd, however, it enables various multiple-round hardware implementations with a small overhead as 36 has many factors.

Block Shuffle. The block shuffle π comes from a result of Suzaki and Minematsu [42]. In [42], it was reported that by changing the block shuffle different from the ordinal cyclic shift one can greatly improve the diffusion of Type-2 GFS. Here, goodness-of-diffusion is measured by the minimum number of rounds that diffuses any input sub-block difference to all output sub-blocks, called DRmax. Smaller DRmax means a faster diffusion. DRmax of cyclic shift with k sub-blocks is k, while there exist shuffles with DRmax $= 2\log_2 k$, called "optimum block shuffle" [42]. Our π is such one[3] with $k = 16$, hence DRmax $= 8$ while DRmax $= 16$ for the cyclic shift. DRmax is connected to the resistance against various attacks. For example, Type-2 GFS with 16 sub-blocks has 33-round impossible differential characteristics and 32-round saturation characteristics. However, using π of Algorithm 2.1 they can be reduced to 14 and 15 rounds.

There exist multiple optimum block shuffles [42]. Hence π was chosen considering other aspects which is not (directly) related to DRmax. In particular, we

[2] We also would like to point out that the security evaluation of LBlock is insufficient. We already found a saturation attack against 22-round LBlock without considering the key schedule, thus the security margin is smaller than the claimed by the authors (20-round), though a recent work [25] shows a 21-round impossible differential attack.

[3] More precisely, an isomorphic shuffle to one presented at Appendix B ($k = 16$, No. 10) of [42].

chose π considering the the number of differentially and linearly active S-boxes (See Table 1 in Section 4).

S-Box. The 4-bit S-box is chosen to satisfy (1) the maximum differential and linear probabilities are 2^{-2}, which is theoretically the minimum for invertible S-box, and (2) the Boolean degree is 3, and (3) the interpolation polynomial contains many terms and has degree 14. Following the AES S-box design, we use a Galois field inversion. Specifically our S-box is defined as $y = S(x) = f((x \oplus b)^{-1})$, where a^{-1} denotes the inverse of a in GF(2^4) (the zero element is mapped to itself.) with irreducible polynomial $z^4 + z + 1$, and $b = 1$ is a constant, and $f(\cdot)$ is an affine function such that $y = f(x)$ with $y = (y_0 \| y_1 \| y_2 \| y_3)$ and $x = (x_0 \| x_1 \| x_2 \| x_3)$ is determined as $y_0 = x_2 \oplus x_3$, $y_1 = x_0 \oplus x_3$, $y_2 = x_0$, and $y_3 = x_1$.

Key Schedule. The key schedule of TWINE enables on-the-fly operations and produces each round key via sequential update of a key state, that is, there is no intermediate key. As mentioned, it uses no bit permutation. As hardware efficiency is not our ultimate goal, the design is rather conservative compared to the recent hardware-oriented ones [12,34,40], yet quite simple. For security, we want our key schedule to have sufficient resistance against slide, meet-in-the-middle, and related-key attacks.

4 Security Evaluation

4.1 Overview

We examined the security of TWINE against various attacks. Due to the page limit, we here focus on the impossible differential and saturation attacks and explain the basic flows of these attacks since they are the most critical attacks in our evaluation. The results on other attacks, such as differential and linear attacks, will also be briefly described.

In this section, we use the notations X^i_j and RK^i_j following Algorithm 2.1, and define $F^i_j(x) \overset{\mathrm{def}}{=} S(\mathrm{RK}^i_j \oplus x)$ for $i = 1, \ldots, 36$, $j = 0, \ldots, 15$, and denote $F^i_j(x) \oplus F^i_j(x \oplus \delta)$ by $F^i_j(\delta)$. For any symbol S let $\bar{\mathsf{S}}^k$ denote the sequence of k symbols, e.g. $\bar{0}^3$ means $(0, 0, 0)$ and $\bar{A}^3$ means (A, A, A).

4.2 Impossible Differential Attack

Generally, impossible differential attack [3] is one of the most powerful attacks against Feistel and GFS-based ciphers, as demonstrated by (e.g.) [14,31,43]. We searched impossible differential characteristics (IDCs) using Kim et al.'s method [21], and found 64 14-round IDCs

$$(0,\alpha_0,0,\alpha_1,0,\alpha_2,0,\alpha_3,0,\alpha_4,0,\alpha_5,0,\alpha_6,0,\alpha_7) \overset{14r}{\not\to} (\beta_0,0,\beta_1,0,\beta_2,0,\beta_3,0,\beta_4,0,\beta_5,0,\beta_6,0,\beta_7,0), \quad (1)$$

where all variables are 4-bit, $\alpha_i \neq 0$, $\beta_j \neq 0$ for some $i, j \in \{0, \ldots, 7\}$ and others are 0. Based on this we can attack against 23-round TWINE-80, where

IDC of 5-th to 18-th rounds with $\alpha_0 \neq 0$ and $\beta_4 \neq 0$ is used, and tries to recover the subkeys of the first 4 rounds and last 5 rounds (144 bits in total). These subkey bits are uniquely determined via its 80-bit subsequence. A similar attack is possible against 24-round TWINE-128, using the IDC with $\alpha_3 \neq 0$ and $\beta_2 \neq 0$.

The outline of our attack against 23-round TWINE-80 is as follows.

Data Collection. We call a set of 2^{32} plaintexts a *structure* if its i-th sub-blocks are fixed to a constant for all $i = 2, 4, 5, 6, 7, 8, 9, 14 \in \{0, \ldots, 15\}$ and the remaining 8 sub-blocks take all 2^{32} values. Suppose we have one structure. From it we extract plaintext pairs having the difference

$$(p_1, p_2, 0, p_3, \bar{0}^6, p_4, p_5, p_6, p_7, 0, p_0), \text{ where } p_i \in \{0,1\}^4 \text{ is non-zero.} \tag{2}$$

We want the 4-round output pairs to be compliant with the left hand side of Eq. (1) with $\alpha_0 \neq 0$ and other α_is being zero. Hence plaintext pairs having no chance to do that are discarded. Here, the property of S-box shows that for any non-zero p_x, $F^i_j(p_x)$ is one of 7 possible values, depending on RK^i_j and p_x. Using this property we identify $2^{54.56}$ plaintext pairs of Eq. (2) that have a chance. Then we encrypt such plaintext pairs and search the ciphertext pairs having the difference

$$(0, c_1, 0, c_2, c_3, c_4, c_0, c_5, c_6, c_7, c_8, c_9, c_{10}, c_{11}, 0, 0), \tag{3}$$

where all c_is are non-zero 4-bit values. We prepare $2^{29.55}$ structures and obtain $2^{68.11}$ ciphertext pairs of the difference Eq. (3) out of all $2^{84.11}$ ciphertext pairs.

Key Elimination. For each ciphertext pair satisfying Eq. (3), we try to eliminate the wrong guesses for the 80-bit (sub)key vector $(\mathcal{K}_1 \| \mathcal{K}_2 \| \mathcal{K}_3)$, where $|\mathcal{K}_1| = 20$, $|\mathcal{K}_2| = 52$, $|\mathcal{K}_3| = 8$ and $\mathcal{K}_1 = (\mathrm{RK}^1_{[1,2,3,7]}, \mathrm{RK}^{23}_0)$, $\mathcal{K}_2 = (\mathrm{RK}^1_{[0,5,6]}, \mathrm{RK}^2_{[2,4,6,7]}, \mathrm{RK}^{23}_{[2,4,5]}, \mathrm{RK}^{22}_{[1,3,4]})$ and $\mathcal{K}_3 = (\mathrm{RK}^{22}_{[0,2]})$ (here $\mathrm{RK}^i_{[a,b,c]}$ denotes $\mathrm{RK}^i_a \| \mathrm{RK}^i_b \| \mathrm{RK}^i_c$). First, we guess $\mathcal{K}_1$ (which can take all possible values). After $\mathcal{K}_1$ is guessed, the number of each 4-bit subkey candidates in $\mathcal{K}_2$ is $(2 \cdot 6 + 4)/7 \approx 2.28$ on average from the property of S-box mentioned above. Once $\mathcal{K}_1$ and $\mathcal{K}_2$ have been fixed, each RK^i_j in $\mathcal{K}_3$ will have $(2 \cdot 6 + 4)/15 \approx 1.07$ candidates, as we have no restrictions on the input difference for Fs relating to these subkeys. From this observation, we expect to eliminate $2^{20} \cdot 2.28^{13} \cdot 1.07^2 \approx 2^{35.69}$ candidates from a set of 2^{80} values for each plaintext-ciphertext pair. In other words, the wrong subkey is eliminated with probability $2^{-44.31}$.

Consequently, we can attack 23-round TWINE-80 with the data complexity $2^{29.55} \cdot 2^{32} = 2^{61.55}$ blocks, the time complexity $2^{84.56} \cdot 22/(23 \cdot 8) = 2^{77.04}$ encryptions, and the memory complexity $2^{80}/64 = 2^{74}$ blocks.

In a similar manner, we can attack 24-round TWINE-128 with the data, time and memory complexity being $2^{52.21}$ blocks, $2^{115.10}$ encryptions and 2^{118} blocks respectively.

4.3 Saturation Attack

Saturation attack [16] is also a powerful attack against GFS-based ciphers. The attack traces the set of variables $(\mathsf{S}_0, \ldots, \mathsf{S}_{15})$, where S_k denotes the saturation status of k-th nibble which is one of the followings:

$$\begin{array}{llll} \textbf{Constant } (C) : \forall i,j, \;\; X_i = X_j & \textbf{All } (A) & : \forall i \neq j, \; X_i \neq X_j \\ \textbf{Balance } (B) \;\; : \bigoplus_i X_i = 0 & \textbf{Unknown } (U) & : \text{Others} \end{array}$$

Let $\alpha = (\alpha_0, \ldots, \alpha_{15})$ and $\beta = (\beta_0, \ldots, \beta_{15})$, $\alpha_i, \beta_i \in \{C, A, B, U\}$, be the initial and the t-round states. If we have $\alpha_i = A$ and $\beta_j \neq U$ for some i and j with probability 1 (i.e. for all keys), $\alpha \overset{tr}{\to} \beta$ is said to be an t-round saturation characteristic (SC). TWINE has 15-round SC with α consisting of one C and fifteen As and β contains 4 Bs (the remainings are U), for example;

$$(\bar{A}^{12}, C, \bar{A}^{3}) \overset{15r}{\to} (\bar{U}^3, B, \bar{U}^5, B, \bar{U}^3, B, U, B), \text{ and} \tag{4}$$

$$(\bar{A}^{6}, C, \bar{A}^{9}) \overset{15r}{\to} (U, B, \bar{U}^3, B, U, B, \bar{U}^3, B, \bar{U}^4). \tag{5}$$

Suppose we use SC of Eq. (5) to break 22-round TWINE-80. We recover 108-bit subkey. From the key schedule, the actual subkey bits needed to be guessed are 72 bits. First we encrypt a set of 2^{60} plaintexts (called S-structure) induced from the left hand side of Eq. (5), and obtain a set of 2^{60} ciphertexts. Now X^i_j has 2^{60} variations for each i, j, and we let $\oplus X^i_j$ to denote the sum of these 2^{60} variations. We also define $F^i_j out$ as $F^i_j(X^i_j)$ and define $\oplus F^i_j out$ analogously. Next we calculate $\oplus X^{17}_0$ and $\oplus F^{16}_0 out$ for each 108-bit subkey candidate. Here X^{17}_0 is uniquely determined by a certain 40 subkey bits (out of 108 bits). Similarly $F^{16}_0 out$ is determined by a certain 60 subkey bits, and the intersection is 28 bits (thus we need 72-bit search). The computation of $\oplus F^{16}_0 out$ requires $2^{73.80}$ F evaluations (amount to $2^{66.34}$ encryptions of 22-round TWINE). For any subkey guess if $\oplus X^{17}_0$ equals to $\oplus F^{16}_0 out$ the saturation status of $\oplus X^{16}_1$ is B. If not, then the guess is wrong and thus eliminated. As this elimination is expected to occur with probability $1 - 1/2^4$, we can reduce the number of subkey candidates from 2^{72} to 2^{68} for one S-structure. With additional 8-bit key guess, the master key is recovered. Summarizing, the attack with an S-structure requires 2^{60} plaintexts to be encrypted, and 2^{77} (which follows from $2^{66.34} + 2^{76} + \rho$, where ρ denotes the computation of X^{17}_0, which is negligible) encryptions. We can further reduce the time complexity by using multiple S-structures. Using 4 structures, we can attack 22-round TWINE-80 with the data, time and memory complexity being 2^{62} blocks, $2^{68.43}$ encryptions and 2^{67} blocks respectively.

In a similar manner (using SC of Eq, (5)), we can attack 23-round TWINE-128 with the data, time and memory complexity being $2^{62.81}$ blocks, $2^{106.14}$ encryptions and 2^{103} blocks respectively.

4.4 Differential / Linear Cryptanalysis

The security against differential cryptanalysis (DC) [4] and linear cryptanalysis (LC) [28] are typically evaluated by the number of differentially and linearly active S-boxes, denoted by AS_D and AS_L, respectively. We performed a computer-based search for differential and linear paths, and evaluated AS_D and AS_L for each round. As a result, the numbers of AS_D and AS_L are the same (Table 1). Since our S-box has 2^{-2} maximum differential and linear probabilities, the maximum differential and linear characteristic probabilities are both 2^{-64} for 15 rounds. Examples of 14-round differential (Δ) and linear (Γ) characteristics having the minimum I/O weights are as follows. Here, 1 denotes an arbitrary non-zero difference (mask) and 0 denotes the zero difference (mask) for Δ (Γ). They involve 30 active S-boxes, and thus the characteristic probability is 2^{-60}.

$$\begin{aligned}\Delta &= (\bar{0}^9, 1, 0, 1, 0, 1, 0, 0) \overset{14r}{\rightarrow} (\bar{0}^3, 1, \bar{0}^4, 1, 0, 0, 1, 0, 0, 1, 1),\\ \Gamma &= (\bar{0}^6, 1, 1, \bar{0}^3, 1, 0, 0, 1, 1) \overset{14r}{\rightarrow} (\bar{0}^9, 1, \bar{0}^3, 1, 0, 1).\end{aligned} \tag{6}$$

Compared to the impossible differential attack, we naturally expect the key recovery attacks exploiting the key schedule with these differential or linear characteristic are less powerful, since they have larger weight (number of non-zero variables) than that of 14-round IDC (having weight 2) and fewer weights imply the more attackable rounds in the key guessing.

We also remark that a computer-based search for the maximum differential probability (rather than the characteristic probability) of GFS was performed by [29]. However, applying their algorithm to our 16-block case seems computationally infeasible.

Table 1. List of differentially and linearly active S-boxes

Round	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
AS_D, AS_L	0	1	2	3	4	6	8	11	14	18	22	24	27	30	32	35	36	39	41	44

4.5 Key Schedule-Based Attacks

Related-Key Differential Attacks. The related-key attack proposed by Biham [2] works when the adversary can somehow modify the key input, typically insert a key differential. For evaluation of such attack, we implemented the search method by Biryukov et al. [6], which counts the number of active S-boxes for combined data processing and key schedule parts. See [6] for the algorithmic details. We searched 4-bit truncated differential paths. As S-box has maximum differential probability being 2^{-2}, we needed 40 (64) active S-boxes for TWINE-80 (TWINE-128).

The full-search was only computationally feasible for TWINE-80. As a result, the number of active S-boxes reaches 40 for the 22-round. Table 2 shows the search result, where ΔKS, ΔRK, ΔX and AS denote key difference, subkey difference, data difference, and the number of active S-boxes.

Other Attacks. For the slide attack [7], the key schedule of TWINE inserts distinct constants for each round. This is a typical way to thwart slide attacks

Table 2. Truncated differential and its active S-box numbers

Rnd	ΔKS	ΔRK	ΔX	AS	Rnd	ΔKS	ΔRK	ΔX	AS	Rnd	ΔKS	ΔRK	ΔX	AS
1	4D010	A2	A255	0	9	20160	0C	4545	19	17	C0104	80	8191	38
2	D8108	E1	6931	6	10	01604	00	108C	20	18	01041	08	0824	38
3	010C3	08	9896	8	11	16040	58	D840	21	19	10410	42	4202	39
4	10C30	46	4462	9	12	60402	80	A0E2	22	20	04102	00	0081	39
5	0C302	20	2288	10	13	0402C	05	A630	27	21	41020	84	8100	41
6	C3020	94	9411	11	14	C02C0	88	8D39	30	22	10208	41	4124	41
7	30201	40	0968	14	15	02C01	10	5A2E	33					
8	02016	12	1306	15	16	2C010	22	62C3	35					

and hence we consider TWINE is immune to the slide attack. For Meet-In-The-Middle (MITM) attack, we confirmed that the round keys for the first 3 (5) rounds contain all key bits for the 80-bit (128-bit) key case. Thus, we consider it is difficult to mount MITM attack (at least in its basic form) against the full-round TWINE.

5 Implementation

5.1 Hardware

We implemented TWINE on ASIC using a 90nm standard cell library with logic synthesis done by *Synopsys DC Version D-2010.03-SP1-1*. Following [8,12], we used Scan Flip-Flops (FFs). In our library, a D-FF and 2-to-1 MUX cost 5.5 GE and 2.25 GE, and a Scan FF costs 6.75 GE. Hence this technique saves 1.0 GE per 1-bit storage.

The result is shown by Table 3 with a comparison. Note that for some algorithms other than TWINE, the synthesis was not done at 100KHz, hence we estimated the throughput by scaling. Table 4 shows the detail of TWINE-80 round-based implementation, where single round function is computed in a clock. We did not perform a thorough logic minimization of the S-box circuit, which currently costs 30 GEs. The S-box logic minimization can further reduce the size. The figures must be taken with cares, because they depend on the type of FF, technology, library, etc [12]. As suggested by [12], we list Gates/Memory Bit in the table, which denotes the size (in GE) of 1-bit memory device used for the key and states.

For serialized implementation, we employ a shared sbox architecture design where single S-box is repeatedly used in the data processing and the key scheduling. For encryption-only TWINE-80, it achieved $1,011$ GEs. We are still working on it, and the details will be given in the near future.

5.2 Software

We implemented TWINE on Atmel AVR 8-bit micro-controller. The target device is ATmega163, which has 16K bytes Flash, 512 bytes EEPROM and 1,024

Table 3. ASIC implementation results

Algorithm	Function	Block (bit)	Key (bit)	Cycles/ block	Throughput (Kbps@100KHz)	Area (GE†)	Gates / Memory bit	Type
TWINE	Enc	64	80	36	178	1,503	6.75	round
TWINE	Enc+Dec	64	80	36	178	1,799	6.75	round
TWINE	Enc	64	128	36	178	1,866	6.75	round
TWINE	Enc+Dec	64	128	36	178	2,285	6.75	round
TWINE	Enc	64	80	393	16.2	1,011	6.75	serial
PRESENT [38]	Enc	64	80	563	11.4	1,000	n/a	serial
PRESENT [8]	Enc	64	80	32	200	1,570	6	round
AES [30]	Enc	128	128	226	57	2,400	6	serial
mCRYPTON [24]	Enc	64	64	13	492.3	2,420	5	round
SEA [26]	Enc+Dec	96	96	93	103	3,758	n/a	round
HIGHT [20]	Enc+Dec	64	128	34	188.25	3,048	n/a	round
KLEIN [17]	Enc	64	80	17	376.4	2,629	n/a	round
KLEIN [17]	Enc	64	80	271	23.6	1,478	n/a	serial
DES [23]	Enc	64	56	144	44.4	2,309	12.19	serial
DESL [23]	Enc	64	56	144	44.4	1,848	12.19	serial
KATAN [12]	Enc	64	80	254	25.1	1,054	6.25	serial
Piccolo [40]	Enc	64	80	27	237	1,496¶	6.25	round
Piccolo [40]	Enc+Dec	64	80	27	237	1,634¶	6.25	round
Piccolo [40]	Enc	64	80	432	14.8	1,043¶	6.25	serial
Piccolo [40]	Enc+Dec	64	80	432	14.8	1,103¶	6.25	serial
LED [18]	Enc	64	80	1872	3.4	1,040	6/4.67◇	serial
PRINTcipher [22]	Enc	48	80	48	12.5	503⋆	n/a	round

† Gate Equivalent : cell area/2-input NAND gate size (2.82).
¶ Includes a key register that costs 360 GEs; Piccolo can be implemented without a key register if key signal holds while encryption.
◇ Mixed usage of two memory units.
⋆ Hardwired key.

Table 4. Component sizes of TWINE-80 encryption

Data Processing (GE)		Key Scheduling (GE)			
Data register	432	Key register	540	S-box out XOR	16
S-box	240	Round const comp.	2	RC register	33
Round key XOR	64	Round const XOR	12	State register	6
S-box out XOR	64	S-box	60	Others/Control	34
				Total	1503

bytes SRAM. We built the four versions: speed-first, ROM-first (minimizing the consumption), and RAM-first, and the double-block, where two message blocks are processed in parallel. Such an implementation works for parellelizable mode of operations. All versions precompute the round keys, i.e. they do not use an on-the-fly key schedule.

In the speed-first version, two rounds are processed in one loop. This removes the block shuffle between the first and second rounds. RAM load (LD) is faster than ROM load (LPM), hence the S-box and the constants are stored at RAM. The data arrangement is carefully considered to avoid carry in the address computation.

Table 5. Software implementations on ATmega163

Algorithm	Key (bit)	Block (bit)	Lang	ROM (byte)	RAM (byte)	Enc (cyc/byte)	Dec (cyc/byte)	ETput /code†	DTput /code‡
TWINE(speed-first)	80	64	asm	1,304	414	271	271	2.14	2.14
TWINE(ROM-first)	80	64	asm	728	335	2,350	2,337	0.40	0.40
TWINE(RAM-first)	80	64	asm	792	191	2,350	2,337	0.43	0.43
TWINE(double block)	80	64	asm	2,294	386	163	163	2.29	2.29
PRESENT [33]	80	64	asm	2,398	528	1,199	1,228	0.28	0.28
DES [36]	56	64	asm	4,314	n/a	1,079	1,019	0.21	0.22
DESXL [36]	184	64	asm	3,192	n/a	1,066	995	0.29	0.31
HIGHT [36]	128	64	asm	8,836	n/a	307	307	0.36	0.36
IDEA [36]	128	64	asm	596	n/a	338	1,924	4.97	0.87
TEA [36]	128	64	asm	1,140	n/a	784	784	1.11	1.11
SEA [36]	96	96	asm	2,132	n/a	805	805	0.58	0.58
AES [9]	128	128	asm	1,912	432	125	181	3.42	2.35

† Encryption Throughput per code: (1/Enc)/(ROM + RAM) (scaled by 10^6).
‡ Decryption Throughput per code: (1/Dec)/(ROM + RAM) (scaled by 10^6).

Table 5 shows comparison of TWINE and other lightweight block ciphers. We list the (scaled) throughput/code ratio for a performance measure (See Table 5 for the formula), following [37]. AES's performance is still quite impressive, however, one can also observe a good performance of TWINE.

Vector Permutation Instruction. We also implemented TWINE on CPU equipped with a SIMD instruction performing a vector permutation, which we call Vector Permutation Instruction (VPI). Examples of VPI are, `vperm` in Motorola AltiVec, `pshufb` in Intel SSE (SSSE3), and `vtbl` in ARM NEON. The power of VPI was first presented by Hamburg [19] for AES, and then the same technique has been applied to various cryptographic functions, e.g. [1, 10, 11]. However, to the best of our knowledge VPI-based *lightweight* block cipher implementation is not known to date. In our VPI-based code, we transform TWINE into an equivalent form shown by the left of Table 2. This form cyclically invokes 4 different shuffles (called half shuffle) on 8 nibbles. Here, "index of RK" denotes the index of round key, RK, given to the round function (from left to right).

For Intel CPU with SSSE3, we use `pshufb` for block shuffle and S-box, and an encryption round of TWINE is computed using only 6 instructions (see the right of Table 2). Here, the left (right) half of input data is in `xmm0`, (`xmm1`), and `eax` contains the address of round key. This implementation is not possible for LBlock due to the use of multiple S-boxes. We remark that this code can treat two blocks at once (which we call double-block code), since each nibble data is stored in a byte structure and XMM registers are 128-bit.

Table 6 shows the result, where x/y denotes x encryption speed and y decryption speed in cycles per byte. We also implemented VPI-based AES [19] and (popular) T-table AES and measured their performance figures. We observe single-block TWINE is comparable to VPI-based AES, and double-block TWINE is even faster. The key schedule for 80-bit (128-bit) key spends about 200 (290) cycles on Core i7 2600S.

round	index of RK	half shuffle
$4i+1$	$0,1,2,3,4,5,6,7$	$[1,0,4,5,2,3,7,6]$
$4i+2$	$0,2,6,4,3,1,5,7$	$[5,3,7,1,6,0,4,2]$
$4i+3$	$0,6,5,3,4,2,1,7$	$[6,7,3,2,5,4,0,1]$
$4i+4$	$0,5,1,4,3,6,2,7$	$[2,4,0,6,1,7,3,5]$

```
movdqa xmm2,[eax]   : load RK
pxor   xmm2,xmm0    : ⊕ RK
movdqa xmm3,[sbox]  : load S-box
pshufb xmm3,xmm2    : apply S-box
pxor   xmm1,xmm3    : ⊕ S-box out
pshufb xmm0,[sh]    : half shuffle
```

Fig. 2. (Left) 4-round structure for SIMD-based implementation. (Right) A code of round function.

Table 6. Enc/Dec speed (in cycles/byte) of TWINE and AES on Intel CPUs

Processor (codename)	TWINE(single)	TWINE(double)	AES(VPI)	AES(T-table)
Core i5 U560 (Arrandale)	9.47 / 9.49	4.77 / 4.77	6.66 / 9.12	14.26 / 19.27
Core i7 2600S (Sandy Bridge)	11.10 / 11.11	5.55 / 5.55	7.42 / 9.44	14.04 / 21.17
Core i3 2120 (Sandy Bridge)	15.06 / 15.06	7.55 / 7.53	10.28 / 12.37	19.03 / 28.68
Xeon E5620 (Westmere-EP)	13.62 / 13.65	6.87 / 6.87	14.72 / 17.82	31.60 / 42.69
Core2Quad Q9550 (Yorkfield)	15.16 / 15.60	7.93 / 7.95	12.16 / 14.39	22.74 / 30.94
Core2Duo E6850 (Conroe)	26.85 / 26.86	14.85 / 14.86	22.04 / 25.82	22.43 / 30.76

6 Conclusions

We have presented a lightweight block cipher TWINE, which has 64-bit block and 80 or 128-bit key. It is primary designed to fit extremely-small hardware, yet provides a notable software performance from micro-controller to high-end CPU. This characteristic mainly originates from the Type-2 generalized Feistel with a highly-diffusive block shuffle. We performed a thorough security analysis, in particular for the impossible differential and saturation attacks. Although the result implies the sufficient security of full-round TWINE, its security naturally needs to be studied further.

Acknowledgments. The authors would like to thank the anonymous reviewers for many valuable comments. We thank Maki Shigeri, Etsuko Tsujihara, Teruo Saito, Takeshi Kawabata, Hiroyasu Kubo and Daisuke Ikemura for discussions and helping implementations.

References

1. Bernstein, D.J., Schwabe, P.: NEON crypto (2012), `http://cr.yp.to/papers.html`
2. Biham, E.: New Types of Cryptanalytic Attacks Using Related Keys. J. Cryptology 7(4), 229–246 (1994)
3. 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)

4. Biham, E., Shamir, A.: Differential cryptanalysis of the data encryption standard. Springer, London (1993)
5. Biryukov, A. (ed.): FSE 2007. LNCS, vol. 4593. Springer, Heidelberg (2007)
6. Biryukov, A., Nikolić, I.: Automatic Search for Related-Key Differential Characteristics in Byte-Oriented Block Ciphers: Application to AES, Camellia, Khazad and Others. In: Gilbert, H. (ed.) EUROCRYPT 2010. LNCS, vol. 6110, pp. 322–344. Springer, Heidelberg (2010)
7. Biryukov, A., Wagner, D.: Slide Attacks. In: Knudsen, L.R. (ed.) FSE 1999. LNCS, vol. 1636, pp. 245–259. Springer, Heidelberg (1999)
8. Bogdanov, A.A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M., 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)
9. Bos, J.W., Osvik, D.A., Stefan, D.: Fast Implementations of AES on Various Platforms. SPEED-CC – Software Performance Enhancement for Encryption and Decryption and Cryptographic Compilers (2009), http://www.hyperelliptic.org/SPEED/
10. Brumley, B.B.: Secure and Fast Implementations of Two Involution Ciphers. Cryptology ePrint Archive, Report 2010/152 (2010), http://eprint.iacr.org/
11. Calik, C.: An Efficient Software Implementation of Fugue. Second SHA-3 Candidate Conference (2010), http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/Aug2010/index.html
12. Cannière, C.D., Dunkelman, O., Knezevic, M.: KATAN and KTANTAN - A Family of Small and Efficient Hardware-Oriented Block Ciphers. In: Clavier, Gaj (eds.) [15], pp. 272–288
13. Canright, D.: A Very Compact S-Box for AES. In: Rao, J.R., Sunar, B. (eds.) CHES 2005. LNCS, vol. 3659, pp. 441–455. Springer, Heidelberg (2005)
14. Chen, J., Jia, K., Yu, H., Wang, X.: New Impossible Differential Attacks of Reduced-Round Camellia-192 and Camellia-256. In: Parampalli, Hawkes (eds.) [32], pp. 16–33
15. Clavier, C., Gaj, K. (eds.): CHES 2009. LNCS, vol. 5747. Springer, Heidelberg (2009)
16. 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)
17. Gong, Z., Nikova, S., Law, Y.W.: KLEIN: A New Family of Lightweight Block Ciphers. In: Juels, A., Paar, C. (eds.) RFIDSec 2011. LNCS, vol. 7055, pp. 1–18. Springer, Heidelberg (2012)
18. Guo, J., Peyrin, T., Poschmann, A., Robshaw, M.J.B.: The LED Block Cipher. In: Preneel, Takagi (eds.) [35], pp. 326–341
19. Hamburg, M.: Accelerating AES with Vector Permute Instructions. In: Clavier, Gaj (eds.) [15], pp. 18–32
20. Hong, D., Sung, J., Hong, S.H., Lim, J.-I., Lee, S.-J., Koo, B.-S., Lee, C.-H., Chang, D., Lee, J., Jeong, K., Kim, H., Kim, J.-S., Chee, S.: HIGHT: A New Block Cipher Suitable for Low-Resource Device. In: Goubin, L., Matsui, M. (eds.) CHES 2006. LNCS, vol. 4249, pp. 46–59. Springer, Heidelberg (2006)
21. Kim, J.-S., Hong, S.H., Sung, J., Lee, S.-J., Lim, J.-I., Sung, S.H.: Impossible Differential Cryptanalysis for Block Cipher Structures. In: Johansson, T., Maitra, S. (eds.) INDOCRYPT 2003. LNCS, vol. 2904, pp. 82–96. Springer, Heidelberg (2003)
22. Knudsen, L.R., Leander, G., Poschmann, A., Robshaw, M.J.B.: PRINTcipher: A Block Cipher for IC-Printing. In: Mangard, Standaert (eds.) [27], pp. 16–32

23. Leander, G., Paar, C., Poschmann, A., Schramm, K.: New Lightweight DES Variants. In: Biryukov (ed.) [5], pp. 196–210
24. Lim, C.H., Korkishko, T.: mCrypton – A Lightweight Block Cipher for Security of Low-Cost RFID Tags and Sensors. In: Song, J.-S., Kwon, T., Yung, M. (eds.) WISA 2005. LNCS, vol. 3786, pp. 243–258. Springer, Heidelberg (2006)
25. Liu, Y., Gu, D., Liu, Z., Li, W.: Impossible Differential Attacks on Reduced-Round LBlock. In: Ryan, M.D., Smyth, B., Wang, G. (eds.) ISPEC 2012. LNCS, vol. 7232, pp. 97–108. Springer, Heidelberg (2012)
26. Mace, F., Standaert, F.X., Quisquater, J.J.: ASIC Implementations of the Block Cipher SEA for Constrained Applications. Proceedings of the Third International Conference on RFID Security (2007), http://www.rfidsec07.etsit.uma.es/confhome.html
27. Mangard, S., Standaert, F.-X. (eds.): CHES 2010. LNCS, vol. 6225. Springer, Heidelberg (2010)
28. Matsui, M.: Linear Cryptanalysis Method for DES Cipher. In: Helleseth, T. (ed.) EUROCRYPT 1993. LNCS, vol. 765, pp. 386–397. Springer, Heidelberg (1994)
29. Minematsu, K., Suzaki, T., Shigeri, M.: On Maximum Differential Probability of Generalized Feistel. In: Parampalli, Hawkes (eds.) [32], pp. 89–105
30. Moradi, A., Poschmann, A., Ling, S., Paar, C., Wang, H.: Pushing the Limits: A Very Compact and a Threshold Implementation of AES. In: Paterson, K.G. (ed.) EUROCRYPT 2011. LNCS, vol. 6632, pp. 69–88. Springer, Heidelberg (2011)
31. Özen, O., Varıcı, K., Tezcan, C., Kocair, Ç.: Lightweight Block Ciphers Revisited: Cryptanalysis of Reduced Round PRESENT and HIGHT. In: Boyd, C., González Nieto, J. (eds.) ACISP 2009. LNCS, vol. 5594, pp. 90–107. Springer, Heidelberg (2009)
32. Parampalli, U., Hawkes, P. (eds.): ACISP 2011. LNCS, vol. 6812. Springer, Heidelberg (2011)
33. Poschmann, A.: Lightweight Cryptography - Cryptographic Engineering for a Pervasive World. Cryptology ePrint Archive, Report 2009/516 (2009), http://eprint.iacr.org/
34. Poschmann, A., Ling, S., Wang, H.: 256 Bit Standardized Crypto for 650 GE - GOST Revisited. In: Mangard, Standaert (eds.) [27], pp. 219–233
35. Preneel, B., Takagi, T. (eds.): CHES 2011. LNCS, vol. 6917. Springer, Heidelberg (2011)
36. Rinne, S.: Performance Analysis of Contemporary Light-Weight Cryptographic Algorithms on a Smart Card Microcontroller. SPEED – Software Performance Enhancement for Encryption and Decryption (2007), http://www.hyperelliptic.org/SPEED/start07.html
37. Rinne, S., Eisenbarth, T., Paar, C.: Performance Analysis of Contemporary Lightweight Block Ciphers on 8-bit Microcontrollers. SPEED-CC – Software Performance Enhancement for Encryption and Decryption and Cryptographic Compilers (2009), http://www.hyperelliptic.org/SPEED/
38. Rolfes, C., Poschmann, A., Leander, G., Paar, C.: Ultra-Lightweight Implementations for Smart Devices – Security for 1000 Gate Equivalents. In: Grimaud, G., Standaert, F.-X. (eds.) CARDIS 2008. LNCS, vol. 5189, pp. 89–103. Springer, Heidelberg (2008)
39. Satoh, A., Morioka, S., Takano, K., Munetoh, S.: A Compact Rijndael Hardware Architecture with S-Box Optimization. In: Boyd, C. (ed.) ASIACRYPT 2001. LNCS, vol. 2248, pp. 239–254. Springer, Heidelberg (2001)
40. Shibutani, K., Isobe, T., Hiwatari, H., Mitsuda, A., Akishita, T., Shirai, T.: Piccolo: An Ultra-Lightweight Blockcipher. In: Preneel, Takagi (eds.) [35], pp. 342–357

41. Shirai, T., Shibutani, K., Akishita, T., Moriai, S., Iwata, T.: The 128-Bit Blockcipher CLEFIA (Extended Abstract). In: Biryukov (ed.) [5], pp. 181–195
42. Suzaki, T., Minematsu, K.: Improving the Generalized Feistel. In: Hong, S., Iwata, T. (eds.) FSE 2010. LNCS, vol. 6147, pp. 19–39. Springer, Heidelberg (2010)
43. Tsunoo, Y., Tsujihara, E., Shigeri, M., Saito, T., Suzaki, T., Kubo, H.: Impossible Differential Cryptanalysis of CLEFIA. In: Nyberg, K. (ed.) FSE 2008. LNCS, vol. 5086, pp. 398–411. Springer, Heidelberg (2008)
44. Wu, W., Zhang, L.: LBlock: A Lightweight Block Cipher. In: Lopez, J., Tsudik, G. (eds.) ACNS 2011. LNCS, vol. 6715, pp. 327–344. Springer, Heidelberg (2011)
45. Zheng, Y., Matsumoto, T., Imai, H.: On the Construction of Block Ciphers Provably Secure and Not Relying on Any Unproved Hypotheses. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 461–480. Springer, Heidelberg (1990)

A Key Schedule for 128-Bit Key

```
Algorithm A.1: TWINE.KeySchedule-128($K_{(128)}, RK_{(32\times 36)}$)
$\mathrm{WK}_{0(4)} \| \mathrm{WK}_{1(4)} \| \dots \| \mathrm{WK}_{30(4)} \| \mathrm{WK}_{31(4)} \leftarrow K$
for $r \leftarrow 1$ to 35
  do
    $\mathrm{RK}^r_{(32)} \leftarrow \mathrm{WK}_2 \| \mathrm{WK}_3 \| \mathrm{WK}_{12} \| \mathrm{WK}_{15} \| \mathrm{WK}_{17} \| \mathrm{WK}_{18} \| \mathrm{WK}_{28} \| \mathrm{WK}_{31}$
    $\mathrm{WK}_1 \leftarrow \mathrm{WK}_1 \oplus S(\mathrm{WK}_0), \mathrm{WK}_4 \leftarrow \mathrm{WK}_4 \oplus S(\mathrm{WK}_{16}),$
    $\mathrm{WK}_{23} \leftarrow \mathrm{WK}_{23} \oplus S(\mathrm{WK}_{30})$
    $\mathrm{WK}_7 \leftarrow \mathrm{WK}_7 \oplus 0 \| CON^r_H, \mathrm{WK}_{19} \leftarrow \mathrm{WK}_{19} \oplus 0 \| CON^r_L$
    $\mathrm{WK}_0 \| \cdots \| \mathrm{WK}_3 \leftarrow Rot4(\mathrm{WK}_0 \| \cdots \| \mathrm{WK}_3)$
    $\mathrm{WK}_0 \| \cdots \| \mathrm{WK}_{31} \leftarrow Rot16(\mathrm{WK}_0 \| \cdots \| \mathrm{WK}_{31})$
$\mathrm{RK}^{36}_{(32)} \leftarrow \mathrm{WK}_2 \| \mathrm{WK}_3 \| \mathrm{WK}_{12} \| \mathrm{WK}_{15} \| \mathrm{WK}_{17} \| \mathrm{WK}_{18} \| \mathrm{WK}_{28} \| \mathrm{WK}_{31}$
$\mathrm{RK}_{(32\times 36)} \leftarrow \mathrm{RK}^1 \| \mathrm{RK}^2 \| \dots \| \mathrm{RK}^{35} \| \mathrm{RK}^{36}$
```

B Test Vectors (in the Hexadecimal Notation)

key length	80-bit	128-bit
key	00112233 44556677 8899	00112233 44556677 8899AABB CCDDEEFF
plaintext	01234567 89ABCDEF	01234567 89ABCDEF
ciphertext	7C1F0F80 B1DF9C28	979FF9B3 79B5A9B8

Recursive Diffusion Layers for (Lightweight) Block Ciphers and Hash Functions

Shengbao Wu[1,2], Mingsheng Wang[3], and Wenling Wu[3]

[1] Institute of Software, Chinese Academy of Sciences, Beijing 100190, P.O. Box 8718, China
[2] Graduate School of Chinese Academy of Sciences, Beijing 100190, China
[3] State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
mingsheng_wang@yahoo.com.cn,
{wushengbao,wwl}@is.iscas.ac.cn

Abstract. Diffusion layers with maximum branch numbers are widely used in block ciphers and hash functions. In this paper, we construct recursive diffusion layers using Linear Feedback Shift Registers (LFSRs). Unlike the MDS matrix used in AES, whose elements are limited in a finite field, a diffusion layer in this paper is a square matrix composed of linear transformations over a vector space. Perfect diffusion layers with branch numbers from 5 to 9 are constructed. On the one hand, we revisit the design strategy of PHOTON lightweight hash family and the work of FSE 2012, in which perfect diffusion layers are constructed by one bundle-based LFSR. We get better results and they can be used to replace those of PHOTON to gain smaller hardware implementations. On the other hand, we investigate new strategies to construct perfect diffusion layers using more than one bundle-based LFSRs. Finally, we construct perfect diffusion layers by increasing the number of iterations and using bit-level LFSRs. Since most of our proposals have lightweight examples corresponding to 4-bit and 8-bit Sboxes, we expect that they will be useful in designing (lightweight) block ciphers and (lightweight) hash functions.

Keywords: Recursive diffusion Layers, linear transformation, branch number, MDS matrix, Linear Feedback Shift Register (LFSR).

1 Introduction

Diffusion layer is one of the core components in a block cipher with confusion layer. And it is also widely used in many other block cipher-based primitives, for instance, hash functions. The choice of a diffusion layer influences both the security and the efficiency of a cryptographic primitive. On the one hand, it plays an important role in providing security against differential cryptanalysis [2] and linear cryptanalysis [12], which are the two most important cryptanalysis of block ciphers. On the other hand, with the same security, an elaborate diffusion layer may lead to a better performance of a cryptographic primitive on hardware or/and software implementation.

The strength of a diffusion layer is usually measured by the notation of branch number. A block cipher using a diffusion layer with a small branch number may suffer unexpected attacks. Therefore, how to construct diffusion layers with big branch numbers and low-cost implementations is a challenge for designers.

The most attractive diffusion layers are those with maximum branch numbers, which are also called perfect or MDS diffusion layers. The common approach to construct them is to extract MDS matrices from MDS codes [11]. Thus, these diffusion layers have matrix representations over $\mathbb{F}_{2^n}$, where n is usually consistent to the bit length of Sbox used in the confusion layer. Many block ciphers [1,14,6,7], especially AES, use this design strategy to construct their diffusion layers.

A problem using MDS matrices as that in AES is that they cannot be implemented in an extremely compact way on hardware. Thus, they are unfitted in resource constrained environments, such as RFID systems and sensor networks. To conquer this drawback while maintain the maximum branch number, a new design strategy was proposed in the document of PHOTON lightweight hash family [9] and then used in designing the diffusion layer of LED lightweight block cipher [8]. Without extracting an MDS matrix in one step, the new strategy constructs a diffusion layer with a bundle-based linear feedback shift register (LFSR)(see Fig.1). That is, in each step, only the last bundle is updated by a linear combination of all of the bundles while other bundles are obtained by shifting the state vector by one position to the left. Each L_i is chosen as a multiplication with an element in $\mathbb{F}_{2^n}$. The LFSR will iterate s times and output the final state. Suppose A is the state transition matrix of LFSR, then the diffusion layer obtained by this strategy is the matrix A^s over $\mathbb{F}_{2^n}$.

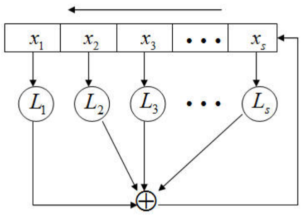


Fig. 1. LFSR for constructing diffusion layers in PHOTON

As mentioned in [9], this design is very compact in hardware implementation because it only needs to realize the LFSR and allows to re-use the existing memory with neither temporary storage nor additional control logic required. Of course, designers would like the final matrix (i.e., A^s) to be MDS, so as to maintain as much diffusion as for the previous strategies. On the other hand, AES-based method (i.e., lookup tables) can be used to implement such cryptographic primitives in software without suffering their efficiency.

In FSE 2012, Sajadieh *et.al* [13] extended this design strategy and proposed a list of perfect diffusion layers. They considered a linear transformation L of vector space $\mathbb{F}_2^n$ and chose $L_i = \sum_{j=-1}^{2} a_i^{(j)} \cdot L^j$, where $a_i^{(j)} \in F_2$ and $1 \leq i \leq s$. The final matrix (i.e., A^s) obtained from this strategy can be treated as an $sn \times sn$ matrix over $\mathbb{F}_2$ or an $s \times s$ matrix composed of linear transformations over $\mathbb{F}_2^n$. To make it perfect, L and $a_i^{(j)}$s should satisfy some conditions. In [13], firstly, the authors studied the sufficient conditions that make a specific diffusion layer with $s = 4$ prefect and then investigated the conditions of other proposals with a necessary statement.

As multiplications with elements in $\mathbb{F}_{2^n}$ are specific linear transformations of vector space $\mathbb{F}_2^n$, the new strategy provides more choices in constructing diffusion layers. Thus, designers may obtain perfect diffusion layers with smaller hardware implementations.

Our Contributions. In this paper, we focus on constructing recursive diffusion layers using LFSRs, following and extending the design strategy of PHOTON and [13]. We construct a list of lightweight perfect diffusion layers with maximum branch numbers from 5 to 9. They mainly distribute in two classes — one class of them are generated by one bundle-based LFSR, using the design strategy of PHOTON and [13], while another class of them are constructed by new strategies using more than one bundle-based LFSRs. Our proposals have smaller hardware implementations than diffusion layers given in PHOTON lightweight hash family and [13]. And they can be used to replace those of PHOTON lightweight hash family. The best replacement can save 22.3% gate equivalents (GE) in the diffusion layer. Finally, we construct perfect diffusion layers by increasing the number of iterations and using bit-level LFSRs.

Outline of This paper. In Section 2, we introduce the definitions of linear transformation, determinant of a matrix over commutative rings and branch number. Previous results on judging perfect diffusion layers are also discussed. Our strategy and some criteria for constructing perfect diffusion layers are described in Section 3. In Section 4 and Section 5, we illustrate our results generated by bundle-based LFSRs. Then, we compare our results with known perfect diffusion layers in Section 6. In Section 7, we investigate some possible manners of constructing new perfect diffusion layers using LFSRs. Finally, we conclude this paper.

2 Preliminaries

In this section, we first introduce the definitions of linear transformation and determinant of a matrix over commutative rings. Then, we introduce the notation of branch number and several statements for constructing prefect diffusion layers.

2.1 Linear Transformation

If V is a vector space over $\mathbb{F}_2$, then a *linear transformation* of V is a map $L : V \to V$ such that

$$L(\mathbf{u} \oplus \mathbf{v}) = L(\mathbf{u}) \oplus L(\mathbf{v}) \tag{1}$$

holds for any $\mathbf{u}, \mathbf{v}$ in V. L is invertible if it is injective and surjective. If linear transformation $L_3 = L_2 \circ L_1$, that is, $L_3(\mathbf{v}) = L_2(L_1(\mathbf{v}))$, then L_3 is invertible if and only if L_1 and L_2 are invertible.

Since there is a square matrix M over $\mathbb{F}_2$ such that $L(\mathbf{v}) = M \cdot \mathbf{v}$, the invertibility of L is equivalent to the non-singularity of M. Thus, in the subsequent discussions, we directly use a matrix to represent a linear transformation. One familiar class of linear transformations is the multiplication with an element in $\mathbb{F}_{2^n}$, that is, $L(\mathbf{v}) = a \cdot \mathbf{v}$, where $a, \mathbf{v} \in \mathbb{F}_{2^n}$. Notice that a can be represented as an $n \times n$ matrix over $\mathbb{F}_2$ if we treat $\mathbf{v}$ as a vector in $\mathbb{F}_2^n$.

2.2 Matrix over Commutative Rings

In this section, we first review several statements of matrix theorem which are true over any commutative ring $\mathcal{R}$. More information is advised to [4]. Then, we introduce a specific commutative ring which is used in this work.

Similar to the classical definition of the determinant, we have

Definition 1. *[4] Let $A = (A_{i,j})_{1\le i\le s, 1\le j\le s}$ be an $s \times s$ matrix with entries in a commutative ring $\mathcal{R}$. The determinant of A, denoted by $det(A)$, is the following element of $\mathcal{R}$:*

$$det(A) = \sum_{\sigma \in P(s)} sgn(\sigma) A_{1,\sigma(1)} A_{2,\sigma(2)} \cdots A_{s,\sigma(s)},$$

where $P(s)$ denotes the set of all permutations on s letters and $sgn(\sigma) \in \{1, -1\}$ is the sign of $\sigma \in P(s)$.

Then, $det(AB) = det(A)det(B)$ and $det(A^T) = det(A)$. Here, A^T is the transposition of A. Similarly, we have

Theorem 1. *[4] Let $A = (A_{i,j})_{1\le i\le s, 1\le j\le s}$, then A is invertible if and only if $det(A) \in U(\mathcal{R})$, where $U(\mathcal{R})$ is the set of all invertible elements in ring $\mathcal{R}$.*

Now, suppose L is an $n \times n$ non-singular matrix over $\mathbb{F}_2$ and

$$S = \{\sum a_{-i} L^{-i} + a_0 + \sum a_j L^j : i, j \in \mathbb{Z}^+, a_{-i}, a_0, a_j \in \mathbb{F}_2\}$$

is a set which includes all polynomials of L and L^{-1}. Then, the set S together with the addition of $\mathbb{F}_2$ and the multiplication of polynomials, form a commutative ring. We denote it by $\mathbb{F}_2[L, L^{-1}]$. Then, we have

Proposition 1. *Let B be an element of $\mathbb{F}_2[L, L^{-1}]$, then $B \in U(\mathbb{F}_2[L, L^{-1}])$ if and only if B is a $n \times n$ non-singular matrix over $\mathbb{F}_2$.*

Proof. If $B \in U(\mathbb{F}_2[L, L^{-1}])$, then there is a $C \in \mathbb{F}_2[L, L^{-1}]$ such that $BC = I$. Thus, when treat B, C and I as matrices over $\mathbb{F}_2$, the determinant $|B| = 1$, i.e, B is non-singular. On the contrary, if B is non-singular over $\mathbb{F}_2$, then there is a positive integer m such that $B^m = I$ (Notice that $m \leq 2^n - 1$ is a finite integer). The smallest m is called the order of B and can be efficiently computed by methods introduced in [5]. Since $B \in \mathbb{F}_2[L, L^{-1}]$, then $B^{n-1} \in \mathbb{F}_2[L, L^{-1}]$ and it is the inverse of B in $\mathbb{F}_2[L, L^{-1}]$. Thus, B is invertible.

2.3 Branch Number

Suppose $\mathbf{v}$ is a vector with s bundles, i.e., $\mathbf{v} = (v_1, v_2, \ldots, v_s)$, where each $v_i \in \mathbb{F}_2^n$ is a vector over a finite field $\mathbb{F}_2$ with length n. The bundle weight of a vector $\mathbf{v}$, denoted by $w_b(\mathbf{v})$, is equal to the number of non-zero bundles. Then, we have

Definition 2. *[7] The differential branch number of a linear diffusion layer D is given by*

$$\mathcal{B}_d(D) = \min_{\mathbf{v} \neq \mathbf{0}}(w_b(\mathbf{v}) + w_b(D(\mathbf{v}))), \tag{2}$$

where D can be represented as an $sn \times sn$ matrix over $\mathbb{F}_2$ or an $s \times s$ matrix consisting of linear transformations of $\mathbb{F}_2^n$.

Similarly, we can define the linear branch number.

Definition 3. *[7] The linear branch number of a linear diffusion layer D is given by*

$$\mathcal{B}_l(D) = \min_{\mathbf{v} \neq \mathbf{0}}(w_b(\mathbf{v}) + w_b(D^T(\mathbf{v}))), \tag{3}$$

where D^T is the transposition of D.

Theorem 2. *[7] A linear diffusion layer D has a maximum differential branch number if and only if it has a maximum linear branch number.*

For a diffusion layer acting on s bundles, the maximal $\mathcal{B}_d$ and $\mathcal{B}_l$ is $s+1$, known as the singleton bound [11]. And D is called a perfect or MDS diffusion layer if it takes its maximal $\mathcal{B}_d$ and $\mathcal{B}_l$.

2.4 Linear Diffusion Layers with Maximum Branch Numbers

In MDS codes, the most widely used property for constructing an MDS matrix is

Theorem 3. *[11] An $[m, s, d]$ code with generator matrix $G = [I_{s\times s} D_{s\times(m-s)}]$ is an MDS code if and only if every square submatrix of D is non-singular, where D is a matrix over $\mathbb{F}_{2^n}$.*

In the Proposition 3.1 and Proposition 3.2 of [3], Blaum *et.al* showed that Theorem 3 is also valid even we substitute every element of D as a linear transformation of vector space $\mathbb{F}_2^n$. Notice that now

$$D = \begin{pmatrix} D_{1,1} & D_{1,2} & \cdots & D_{1,m-s} \\ D_{2,1} & D_{2,2} & \cdots & D_{2,m-s} \\ \vdots & \vdots & \ddots & \vdots \\ D_{s,1} & D_{s,2} & \cdots & D_{s,m-s} \end{pmatrix} \tag{4}$$

is a block matrix with s rows and $m-s$ columns, where each $D_{i,j}$ is an $n \times n$ matrix over $\mathbb{F}_2$.

We denote by $\mathcal{D}_{i\times j}$ a submatrix obtained from D by deleting $(s-i)$ block rows and $(m-s-j)$ block columns while maintaining the order of other elements in D. When considering a linear diffusion layer, D is a square matrix, that is, $m = 2s$. Then, the results of [3] can be re-described as the following statement.

Theorem 4. *A linear diffusion layer D has a maximum branch number if and only if every square submatrix of D, i.e., $\mathcal{D}_{k\times k}$ for $1 \le k \le s$, is non-singular.*

To detect a perfect linear diffusion layer, we need to judge whether all $\sum_{k=1}^{s}\binom{s}{k}\binom{s}{k} = \binom{2s}{s} - 1$ submatrices of D are non-singular according to Theorem 4.

Remark 1. In the paper [13], Sajadieh *et. al* used the necessary part of this statement to describe the conditions of some perfect diffusion layers. Now, we know that they are enough to make those diffusion layers perfect.

3 Our Strategy for Constructing Diffusion Layers

In this paper, we will construct perfect diffusion layers using different kinds of LFSRs. The strategy introduced in this section are suitable for bundle-based LFSRs, that is, one or several bundles are updated in each step while others are obtained by the shift operation. The procedure has four steps.

1. Construct an $s \times s$ matrix $A = (A_{i,j})_{1\le i,j\le s}$ with each $A_{i,j} = \sum a_k^{(i,j)} \cdot L^k \in \mathbb{F}_2[L, L^{-1}]$. Of course, matrix A will be chosen with some structures for low-cost hardware implementations and reducing the search space.
2. Choose an integer d and compute $D = A^d$ $(d \ge 1)$ as the final diffusion layer. Since D is a matrix over $\mathbb{F}_2[L, L^{-1}]$, from Theorem 1 and Theorem 4, we deduce that D is perfect if and only if the determinant of each square submatrix of D is an invertible element in $\mathbb{F}_2[L, L^{-1}]$.
3. Generate the determinants of all square submatrices of D (i.e., $\mathcal{D}_{k\times k}$ for $1 \le k \le s$) as the conditions, which is a set of polynomials in $\mathbb{F}_2[L, L^{-1}]$. Notice that if zero is in the condition set, we know D can not be MDS. In this case, we will change the choice of A or d.
4. Search whether there exists any L such that all polynomials obtained in step 3 are invertible elements in $\mathbb{F}_2[L, L^{-1}]$. Based on Proposition 1, we need to check whether all conditions are non-singular matrix over $\mathbb{F}_2$.

The procedure above can be performed systematically on a computer. To find perfect diffusion layers with low-cost faster, several criteria are used in this paper.

1. Choose $A_{i,j}$ with few terms. That is, the number of 1's in the coefficient list $[\ldots, a_{-1}^{(i,j)}, a_0^{(i,j)}, a_1^{(i,j)}, \ldots]$ should be as few as possible. The degree of L and L^{-1} are also chosen to be low. Thus, $A_{i,j}$ may be chosen as 0, that is, zero transformation.

2. The integer d should be chosen as small as possible, since it determines the efficiency of getting the final diffusion layer D in both hardware and software implementation.
3. The linear transformation L should be low-cost in hardware implementation. Our chief targets are linear transformations with no more than one XOR gate (Note: one XOR gate needs about 2.66 GE in hardware implementation). Multiplications with elements in $\mathbb{F}_{2^n}$ will be the secondary choices.

Additionally, for practical applications, we expect that

4. The perfect diffusion layers proposed in this paper should have examples for $n = 4$ and $n = 8$, since 4-bit Sboxes and 8-bit Sboxes are involved in many cryptographic primitives.

Remark 2. Suppose $\mathbf{y} = L{\cdot}\mathbf{x}$, where L is an invertible matrix and $\mathbf{x}, \mathbf{y}$ are column vectors in $\mathbb{F}_2^n$. A linear transformation without any XOR gate is a permutation of the input bits, that is, $\mathbf{y}_i = \mathbf{x}_{i'}$, where $[1', 2', \ldots, n']$ is a permutation of $[1, 2, \ldots, n]$. Thus, L is a matrix with exactly n nonzero entries, satisfying that each row and each column of L have exactly one nonzero entry. Similarly, a linear transformation with only one XOR gate is a matrix with exactly $n + 1$ nonzero entries, satisfying (1) each row and each column have at least one nonzero entries and (2) there exists a unique row that has two nonzero entries.

All choices of L with no more than one XOR gate is $n! + n! \cdot (n^2 - n)$. For $n = 4$ and $n = 8$, we may enumerate all of them efficiently. For $n \geq 16$, only a small part of them can be enumerated. In this paper, we fix $L[i, i + 1] = 1$ (for $1 \leq i \leq n - 1$) and $L[n, 1] = 1$, and the search space is reduced to $n^2 - n + 1$.

In the subsequent two sections, we illustrate our results in constructing perfect diffusion layers using bundle-based LFSRs. One class of them are obtained by iterating one LFSR several times, following the design strategy of PHOTON and [13]. In this design, only one bundle is updated while others are obtained by shifting the state vector by one position to the left. Then, we extend the strategy to find them using several bundle-based LFSRs in an iteration.

4 Construct Perfect Diffusion Layers with One Bundle-Based LFSR

In this section, we revisit the design strategy of PHOTON and [13], which constructs recursive diffusion layers with one bundle-based LFSR. That is, we try to construct its state transition matrix A and choose an iteration number d such that $D = A^d$ is perfect, where

$$A = \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ L_1 & L_2 & L_3 & \cdots & L_s \end{pmatrix} \tag{5}$$

and $L_i = \bigoplus a_k^{(i)} \cdot L^k$. For simplicity, we only extract the final row of A, that is,

$$A_{lfsr}^{(s)} = [L_1, L_2, \ldots, L_s],$$

to illustrate our choice of the LFSR. The cost of A in hardware implementation is $(s-1)n + \sum_{i=1}^{s} \#L_i$ XOR gates if all L_is are nonzero elements of $\mathbb{F}_2[L, L^{-1}]$, for which $\#L_i$ XOR gates are allocated to the linear transformation L_i. To compare our results with those given in LED, PHOTON and [13], we limit $d \leq s$ in this section.

4.1 Perfect Diffusion Layers for $s = 4$

The best result we find for $s = 4$ is

$$A_{lfsr}^{(4)} = [L, 1, 1, L^2] \tag{6}$$

with $d = 4$. It costs $3n + \#L + \#L^2$ XOR gates in hardware implementation.

For the convenience of comprehension, we use this example to display how to generate a condition set. Other proposals in this paper are done similarly.

Following the strategy discussed in Section 3, once $A_{lfsr}^{(4)} = [L, 1, 1, L^2]$ and $d = 4$ are chosen, we calculate

$$D = A^4 = \begin{pmatrix} L & 1 & 1 & L^2 \\ L^3 & L^2 + L & L^2 + 1 & L^4 + 1 \\ L^5 + L & L^4 + L^3 + 1 & L^4 + L^2 + L + 1 & L^6 + 1 \\ L^7 + L & L^6 + L^5 + L + 1 & L^6 + L^4 + L^3 & L^8 + L^4 + L + 1 \end{pmatrix}.$$

Now, based on Theorem 4, we need to calculate the determinants of all the square submatrices $\mathcal{D}_{k\times k}$ of D. Suppose F is a determinant of $\mathcal{D}_{k\times k}$ for some k, which is a polynomial in $\mathbb{F}_2[L, L^{-1}]$, it can be factorized as

$$F = F_1^{i_1} \cdot F_2^{i_2} \cdots F_j^{i_j},$$

where $F_1, \ldots, F_j$ are irreducible polynomials and $i_1, \ldots, i_j$ are positive integers. Then, F is non-singular if and only if its factors $F_1, \ldots, F_j$ are non-singular. Thus, $F_1, \ldots, F_j$ are added to the condition set. For example, suppose $k = 1$ and $\mathcal{D}_{1\times 1} = D_{2,4} = L^4 + 1$, then $L + 1$ is added to the condition set since $L^4 + 1 = (L + 1)^4$.

After enumerating the determinants of all 69 square submatrices $\mathcal{D}_{k\times k}$ $(1 \leq k \leq 4)$, we conclude that $D = A^4$ with $A_{lfsr}^{(4)} = [L, 1, 1, L^2]$ has branch number 5, if the following 12 matrices

$$\begin{array}{ccc} L, & L+1, & L^2+L+1, \\ L^3+L+1, & L^3+L^2+1, & L^4+L^3+1, \\ L^4+L^3+L^2+L+1, & L^5+L^2+1, & L^5+L^4+L^3+L+1, \\ L^6+L^5+L^4+L+1, & L^6+L^5+L^4+L^2+1, & L^7+L^6+L^5+L^4+1 \end{array}$$

are non-singular.

Table 1. Lightweight linear transformations L for $A_{lfsr}^{(4)} = [L, 1, 1, L^2]$

Length of the input	example of L
$n = 4$	$[[2,3],3,4,1]$
$n = 8$	$[[5,6],7,5,8,4,3,1,2]$
$n = 16$	$[[1,2],3,4,\ldots,16,1]$
$n = 32$	$[[2,4],3,4,\ldots,32,1]$
$n = 64$	$[[2,6],3,4,\ldots,64,1]$

Finally, we introduce some lightweight linear transformations of $\mathbb{F}_2^n$ that satisfy the above conditions (see Table 1). Each of them costs only one XOR gate (2.66 GE) in hardware implementation. Thus, L^2 costs two XOR gates. Note that there are many other similar linear transformations. For simplicity, we extract the nonzero positions in each row of a matrix to represent it. For example,

$[[2,3],3,4,1]$ is the representation of matrix $\begin{pmatrix} 0\,1\,1\,0 \\ 0\,0\,1\,0 \\ 0\,0\,0\,1 \\ 1\,0\,0\,0 \end{pmatrix}$.

Other Information from the Condition Set. From the condition set, we observe that $L^4 + L + 1$ does not belong to it. Thus, for $n = 4$, L can be chosen as the multiplication with α, i.e., $L(v) = \alpha \cdot v$ ($v \in \mathbb{F}_{2^4}$), where α is a root of the irreducible polynomial $x^4 + x + 1$. This L also costs one XOR gate in hardware implementation [8].

A question is that why the multiplication with α is also a valid choice. That is, after replacing L by the multiplication with α, can we make sure that all 12 conditions in the condition set are invertible elements of $\mathbb{F}_{2^4}$? The answer is yes and the reasons are shown as following.

– $1, \alpha, \alpha^2$ and α^3 compose a basis of $\mathbb{F}_{2^4}$, since α is a root of the irreducible polynomial $x^4 + x + 1$. That is, for each $\beta \in \mathbb{F}_{2^4}$, there is a unique vector $[a_0, a_1, a_2, a_3] \in \mathbb{F}_2^4$ such that $\beta = a_0 + a_1\alpha + a_2\alpha^2 + a_3\alpha^3$.
– Suppose $g(x) \neq x^4 + x + 1$ is another irreducible polynomial, then

$$g(\alpha) \equiv a_0 + a_1\alpha + a_2\alpha^2 + a_3\alpha^3 \bmod \alpha^4 + \alpha + 1$$

is a nonzero element. Thus, $g(\alpha)$ is invertible. For a further step, all conditions in the condition set are irreducible polynomials and not equal to $x^4 + x + 1$, which implies that they are invertible elements of $\mathbb{F}_{2^4}$ if L is chosen as the multiplication with α.

In general, if we observe that an irreducible polynomial $f(x) = x^n + \phi(x)$ does not belong to the condition set, then the multiplication with one of its roots can be chosen as a candidate of L to obtain a perfect diffusion layer over $\mathbb{F}_{2^n}$.

Table 2. Perfect diffusion layers we find for $5 \le s \le 8$, together with the cost in hardware implementation and the number of conditions

	$A_{lfsr}^{(s)}$ $(d = s)$	Cost (XOR gates)	No. of Cond.
$s = 5$	$[1, L^2, L^{-1}, L^{-1}, L^2]$	$4n + 2(\#L^2 + \#L^{-1})$	21
$s = 6$	$[1, L^{-2}, L^{-1}, L^2, L^{-1}, L^{-2}]$	$5n + \#L^2 + 2(\#L^{-2} + \#L^{-1})$	90
$s = 7$	$[1, L, L^{-5}, 1, 1, L^{-5}, L]$	$6n + 2(\#L + \#L^{-5})$	592
$s = 8$	$[1, L^{-3}, L, L^3, L^2, L^3, L, L^{-3}]$	$7n + \#L^2 + 2(\#L + \#L^3 + \#L^{-3})$	2629

Table 3. Lightweight linear transformations L for $A_{lfsr}^{(s)}$ with $5 \le s \le 8$

Length of the input	example of L	fit for
$n = 4$	$[[2, 3], 3, 4, 1]$	$s = 5, 6, 7, 8$
$n = 8$	$[[5, 6], 7, 5, 8, 4, 3, 1, 2]$	$s = 5, 6, 7, 8$
$n = 16$	$[[1, 2], 3, 4, \dots, 16, 1]$	$s = 5, 6$
$n = 16$	$[[2, 6], 3, 4, \dots, 16, 1]$	$s = 7, 8$
$n = 32$	$[[2, 10], 3, 4, \dots, 32, 1]$	$s = 5, 6, 7, 8$
$n = 64$	$[[2, 3], 3, 4, \dots, 64, 1]$	$s = 6, 7$
$n = 64$	$[[2, 17], 3, 4, \dots, 64, 1]$	$s = 5, 8$

Table 4. Comparison of our diffusion layers with those used in PHOTON

	P_{100}	P_{144}	P_{196}	P_{256}	P_{288}
(s, n)	$(5, 4)$	$(6, 4)$	$(7, 4)$	$(8, 4)$	$(6, 8)$
PHOTON	75.33 GE	80 GE	99 GE	145 GE	144 GE
Ours	58.52 GE	74.48 GE	95.76 GE	117.04 GE	127.68 GE
Reduced(%)	22.3	6.9	3.3	19.3	11.3

4.2 Results for $5 \le s \le 8$

The best results we find for $5 \le s \le 8$ are listed in Table 2. In this table, we also list their cost in hardware implementation. With the increment of s, the number of conditions that must be satisfied increases rapidly. Due to the lack of space, we only introduce the conditions for $s = 5$ in Appendix A.

Several lightweight examples of these diffusion layers are given in Table 3. All of them and their inverses only cost one XOR gate in hardware implementation. An interesting observation is that $L^4 + L + 1$ is not included in any of these condition sets. Thus, the multiplication with α is also a choice for $n = 4$, where α is a root of irreducible polynomial $x^4 + x + 1$.

Application. PHOTON lightweight hash family has 5 variants according to the size of its internal permutation. Our perfect diffusion layers can be used to substitute those of PHOTON and obtain smaller hardware implementations. The specification is given in Table 4. Note that our diffusion layers may perform better in practice under some available techniques. For instance, in the document

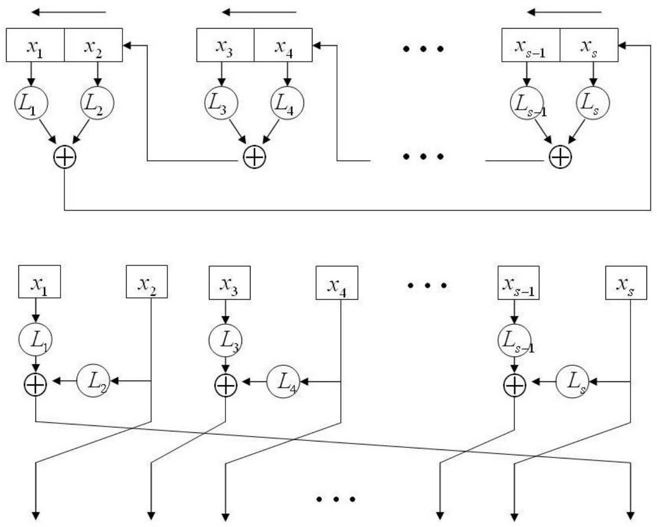


Fig. 2. Several bundle-based LFSRs for constructing diffusion layers

of PHOTON, the authors mentioned that using their library, the multiplications with α, α^2 and α^3 can be implemented in hardware with 2.66 GE, 4.66 GE and 7 GE when using the irreducible polynomial $x^4 + x + 1$, respectively. However, we evaluate them with 2.66 GE, 5.32 GE and 7.98 GE in Table 4, respectively. We would like to remark that the diffusion layer of P_{288} can be further improved using the results of the next section.

5 Construct Perfect Diffusion Layers with Several Bundle-Based LFSRs

In this section, we construct perfect diffusion layers with more than one bundle-based LFSRs. We consider $\frac{s}{2}$ LFSRs (see Fig.2, upper part, s is even) in an iteration, where each LFSR composed of two bundles and they form a head-tail connecting circle. In each step, the last bundle of each LFSR is updated by a linear combination of all of the bundles in the next LFSR while other bundles are obtained by shifting the state vector of each LFSR by one position to the left. Similar to the diffusion layers constructed by one LFSR, this mode also allows to re-use the existing memory with neither temporary storage nor additional control logic required.

From the point of view of block cipher structures, these LFSRs consist of a Type-II Generalized Feistel Structure (GFS, [15]) (see Fig.2, nether part). Thus, to obtain perfect diffusion layers, we firstly construct a matrix

$$A = \begin{pmatrix} T & U_2 & 0 & \cdots & 0 & 0 \\ 0 & T & U_3 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & T & U_{\frac{s}{2}} \\ U_1 & 0 & 0 & \cdots & 0 & T \end{pmatrix}, \tag{7}$$

and then make $D = A^d$ perfect, where "0" is a $2n \times 2n$ zero matrix over $\mathbb{F}_2$ while $T = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$ and $U_i = \begin{pmatrix} 0 & 0 \\ L_{2i-1} & L_{2i} \end{pmatrix}$ $(1 \leq i \leq \frac{s}{2})$ are 2×2 block matrices. We also focus on $L_i = \bigoplus a_k^{(i)} \cdot L^k$ and use

$$A_{gfs}^{(s)} = [L_1, L_2, \ldots, L_s]$$

to indicate our choice of A. The cost of implementing the LFSRs is $\frac{s}{2} \cdot n + \sum_{i=1}^{s} \#L_i$ XOR gates if all L_is are nonzero elements in $\mathbb{F}_2[L, L^{-1}]$.

5.1 Perfect Diffusion Layers for $s = 4$

The best results we find for $s = 4$ is

$$A_{gfs}^{(4)} = [L, 1, 1, L] \tag{8}$$

with $d = 4$, if the following 7 matrices

$$L, \quad L+1, \; L^2+L+1, \; L^3+L+1,$$
$$L^3+L^2+1, \; L^4+L^3+1, \; L^4+L^3+L^2+L+1$$

are non-singular.

It costs $2n+2\#L$ XOR gates in hardware implementation. Since the condition set of this choice is included in that of $A_{lfsr}^{(4)} = [L, 1, 1, L^2]$ with $d = 4$, all examples listed in Table 1 and the multiplication with a root of the irreducible polynomial $x^4 + x + 1$ for $n = 4$ fit the above seven conditions.

5.2 Results for $s = 6$ and $s = 8$

The best results we find for $s = 6$ is

$$A_{gfs}^{(6)} = [L, 1, 1, L^2, L, L^2] \tag{9}$$

with $d = 6$. It costs $3n + 2\#L + 2\#L^2$ XOR gates in hardware implementation. And 196 conditions need to be satisfied. However, we do not find linear transformations with no more than one XOR gates when $n = 4$ and $n = 8$. For $n = 4$, we find all irreducible polynomials with degree 4 are included in the condition set. Thus, there is no choices for $L \in \mathbb{F}_{2^4}$. For $n = 8$, we find four irreducible polynomials $L^8 + L^6 + L^5 + L^2 + 1$, $L^8 + L^6 + L^3 + L + 1$,

$L^8 + L^6 + L^5 + L^4 + 1$ and $L^8 + L^7 + L^3 + L^2 + 1$ are not included in the condition set. Therefore, L can be chosen as the multiplication with a root of these polynomials, which costs 3 XOR gates in hardware implementation. Using this diffusion layer $(2.66 \times (3 \times 8 + 2 \times 3 + 2 \times 6) = 112$ GE) to substitute that used in P_{288} of PHOTON, we may save 22.2% gate equivalents.

The best results we find for $s = 8$ is

$$A_{gfs}^{(8)} = [1, L^4, 1, L^{-1}, 1, L, 1, L^2] \tag{10}$$

with $d = 8$. It costs $4n + \#L^4 + \#L + \#L^{-1} + \#L^2$ XOR gates in hardware implementation. 8692 conditions need to be satisfied. However, we do not find linear transformations with no more than one XOR gate when $n = 4$ and $n = 8$. What's more, all irreducible polynomials with degree 4 and 8 are included in the condition set. Thus, neither $L \in \mathbb{F}_{2^4}$ nor $L \in \mathbb{F}_{2^8}$ satisfies the condition set.

6 Comparison with Known Results

In this section, we compare our bundle-based proposals with those given in the document of LED, PHOTON and [13]. The comparison mainly consists of two parts — the cost in hardware implementation and low-cost examples for $n \in \{4, 8, 16, 32, 64\}$. Table 5 illustrates the comparison results. In this table, we generalize the choices in the LED and PHOTON hash family, which only considered the examples over $\mathbb{F}_{2^4}$, to check whether they have lightweight examples for $n \geq 4$. And diffusion layers proposed in [13] are also re-calculated under the process of Section 3. "Y" means we find an example with only one XOR gate, "Y_F" means we do not find examples with no more than one XOR gate, but there is an example if L is chosen as the multiplication with an element in $\mathbb{F}_{2^n}$. "N" means we find neither examples with no more than one XOR gate nor examples in $L \in \mathbb{F}_{2^n}$.

Some observations are given as follows.

1. All of our proposals (for $4 \leq s \leq 8$) using one bundle-based LFSR have examples with one XOR gate when $n \in \{4, 8, 16, 32, 64\}$, together with the diffusion layers used in LED, $P_{100}, P_{144}, P_{195}$ and P_{256} of PHOTON. And our proposals have smaller hardware implementation than them.
2. We compare our proposals with 9 diffusion layers given in [13]. We find four of them can not be perfect when $d = s$. Another four of them have bigger hardware implementation than our proposals. Only the diffusion layer $A_{lfsr}^{(5)} = [1, L^2, 1, 1, L]$ with $d = 5$ has slightly better performance than our proposal when $n \geq 16$. However, this diffusion layer does not have examples when $n = 4$ and it does not have examples with no more than one XOR gate when $n = 8$.
3. Diffusion layers with the smallest hardware implementation in Table 5 are those constructed by more than one bundle-based LFSRs, especially for $s = 4$, which has examples with one XOR gate for all $n \in \{4, 8, 16, 32, 64\}$. The case with 8 branches (i.e., $s = 8$) may suffer restrictions in practice because it does not have examples when $n = 4$ and $n = 8$.

Table 5. Comparison of our diffusion layers with those given in the LED, PHOTON hash family, and [13]

s	$A^{(s)}_{lfsr}$ $(d=s)$	Cost (XOR gates)	n=4	n=8	n=16	n=32	n=64	Note
4	$[L,1,1,L^2]$	$3n+\#L+\#L^2$	Y	Y	Y	Y	Y	Ours
	$[L^2,1,L,L]$	$3n+2\#L+\#L^2$	Y	Y	Y	Y	Y	LED [8]
	$[1,L,1,1+L]$	$4n+2\#L$	Y	Y	Y	Y	Y	[13]
5	$[1,L^2,L^{-1},L^{-1},L^2]$	$4n+2(\#L^2+\#L^{-1})$	Y	Y	Y	Y	Y	Ours
	$[1,L,L^3+1,L^3+1,L]$	$6n+2\#L+2\#L^3$	Y	Y	Y	Y	Y	P_{100} [9]
	$[1,L^2,1,1,L]$	$4n+\#L+\#L^2$	N	Y_F	Y	Y	Y	[13]
	$[1,L,1,L,L+L^{-1}]$	Not perfect when $d=5$!	-	-	-	-	-	[13]
6	$[1,L^{-2},L^{-1},L^2,L^{-1},L^{-2}]$	$5n+\#L^2+2(\#L^{-2}+\#L^{-1})$	Y	Y	Y	Y	Y	Ours
	$[1,L,L^3,L^2+1,L^3,L]$	$6n+\#L+\#L^2+2\#L^3$	Y	Y	Y	Y	Y	P_{144} [9]
	$[L,L+1,1,L,1,L^2]$	$6n+2\#L+\#L^2$	N	Y_F	Y_F	Y	Y	P_{288} [9]
	$[1,L^2,L^2,L,L^2,1+L]$	Not perfect when $d=6$!	-	-	-	-	-	[13]
	$[1,L,L^{-1},1+L,L,L+L^{-1}]$	$7n+2\#L^{-1}+4\#L$	N	N	Y	Y	Y	[13]
7	$[1,L,L^{-5},1,1,L^{-5},L]$	$6n+2(\#L+\#L^{-5})$	Y	Y	Y	Y	Y	Ours
	$[1,L^2,L^2+L,1,1,L^2+L,L^2]$	$8n+2\#L+4\#L^2$	Y	Y	Y	Y	Y	P_{196} [9]
	$[1,L^2,1+L^2,L,1+L^2,L^2,L^2]$	$8n+\#L+5\#L^2$	N	N	Y_F	Y	Y	[13]
	$[1,L+L^{-1},L,L^{-1},L,L,L^{-1}+1]$	Not perfect when $d=7$!	-	-	-	-	-	[13]
8	$[1,L^{-3},L,L^3,L^2,L^3,L,L^{-3}]$	$7n+\#L^2+2(\#L+\#L^3+\#L^{-3})$	Y	Y	Y	Y	Y	Ours
	$[L,L^2,L,L^3+L+1,L,L^3,L^2+1,L^2+L]$	$11n+5\#L+3\#L^2+2\#L^3$	Y	Y	Y	Y	Y	P_{256} [9]
	$[1,1+L^2,L,1+L,1,L+L^2,L^2,L^2]$	$10n+3\#L+4\#L^2$	N	N	Y_F	Y	Y	[13]
	$[1,1+L,1,L^{-1},L^{-1}+1,L,L,L^{-1}+L]$	Not perfect when $d=8$!	-	-	-	-	-	[13]
	$A^{(s)}_{gfs}$ $(d=s)$							
4	$[L,1,1,L]$	$2n+2\#L$	Y	Y	Y	Y	Y	Ours
6	$[L,1,1,L^2,L,L^2]$	$3n+2\#L+2\#L^2$	N	Y_F	Y_F	Y	Y	Ours
8	$[1,L^4,1,L^{-1},1,L,1,L^2]$	$4n+\#L^4+\#L+\#L^{-1}+\#L^2$	N	N	Y_F	Y_F	Y	Ours

7 Other Recursive Diffusion Layers

In this section, we discuss some possible manners to construct new lightweight perfect diffusion layers using LFSRs.

7.1 Increase the Number of Iterations

We may obtain more lightweight perfect diffusion layers if the number of iterations is increased. However, the efficiency of getting the final diffusion layer D is decreased.

When constructing perfect diffusion layers with branch number 5 (i.e., $s = 4$) using one bundle-based LFSR (see Fig.1), we find some L_is may be chosen as the zero transformation if the number of iterations can be greater than s. An example we obtained is

$$A^{(4)}_{lfsr} = [1, L, 0, 0]. \tag{11}$$

It costs $n + \#L$ XOR gates in hardware implementation and needs 22 iterations to reach branch number 5, if the following eight matrices

$$\begin{array}{ccc} L, & L+1, & L^2+L+1, \\ L^3+L+1, & L^3+L^2+1, & L^4+L^3+1, \\ L^4+L^3+L^2+L+1, & L^5+L^4+L^3+L^2+1 & \end{array}$$

are non-singular.

We observe that all examples listed in Table 1 and the multiplication with a root of the irreducible polynomial $x^4 + x + 1$ for $n = 4$ fit the above eight conditions.

7.2 Bit-Level LFSRs

Diffusion layers discussed above are constructed by bundle-based LFSRs. An instinctive idea is to construct them using bit-level LFSR. That is, the updating unit in the LFSR is not bundle but bit now. In each step, only a few bits, for instance, the rightmost m bits, of LFSR are updated by the XOR values of chosen bit positions while other $sn-m$ bits are obtained by shifting the LFSR by m bits to the left. When considering bit-level LFSR, Theorem 4 will be directly used to judge whether a choice of LFSR is perfect after some iterations.

We search all possible LFSRs when $s = n = 4$ and $m \in \{1, 2\}$, aim to find perfect diffusion layers with branch 5 under 4-bit Sboxes. We denote by $x[1], x[2], \ldots, x[16]$ the 16 bits in the LFSR (see Fig.1), where $x[16]$ is the least significant (rightmost) bit.

- For $m = 1$, we do not find perfect diffusion layers. However, we find many almost perfect diffusion layers [10], that is, with differential branch number 4. They can be detected by a variant of Theorem 4, which will be introduced in the full version due to the lack of space. Although these diffusion layers do not reach the maximum branch number, they may still have some applications because they are extremely lightweight in hardware implementation. For example, one LFSR we find is: $y = x[1] \oplus x[7], x[i] = x[i+1]$ for $1 \le i \le 15, x[16] = y$, which only costs one XOR gate and needs 44 iterations to reach differential branch number 4. Another example is: $y = x[1] \oplus x[6] \oplus x[13], x[i] = x[i+1]$ for $1 \le i \le 15, x[16] = y$. It costs two XOR gates and needs 15 iterations to reach differential branch number 4.

– For $m = 2$, we find a list of perfect diffusion layers. One of the best LFSR is: $y = x[1] \oplus x[6] \oplus x[8] \oplus x[10] \oplus x[13]$, $z = x[2] \oplus x[5] \oplus x[7] \oplus x[10] \oplus x[11] \oplus x[13] \oplus x[14]$, $x[i] = x[i+2]$ for $1 \le i \le 14$, $x[15] = y$, $x[16] = z$. It costs 10 XOR gates and needs 8 iterations to reach branch number 5.

8 Conclusion

In this paper, we construct a list of lightweight perfect diffusion layers using LFSRs. On the one hand, we revisit the design strategy of PHOTON and [13], which constructs perfect diffusion layers using one bundle-based LFSR. Our proposals have smaller hardware implementations than those given in LED, PHOTON and [13]. They can be used to replace the diffusion layers in PHOTON to gain better performance. On the other hand, we extend the strategy to construct perfect diffusion layers using more than one bundle-based LFSRs. The structure we choose is the Type-II Generalized Feistel Structure. Finally, we discuss some possible manners to construct perfect diffusion layers by increasing the number of iterations and using bit-level LFSRs.

Since most of our proposals have low-cost examples which are consistent with 4-bit Sboxes and 8-bit Sboxes, we expect that they will be useful in designing (lightweight) block ciphers and (lightweight) hash functions.

Acknowledgements. We are grateful to the anonymous reviewers for their valuable comments on this paper. This work was partially supported by the 973 Program (Grant No. 2013CB834203) and the National Natural Science Foundation of China (Grant No. 60970134 and Grant No. 11171323).

References

1. Barreto, P.S.L.M., Rijmen, V.: The Anubis block cipher. NESSIE (September 2000), (primitive submitted) http://www.cryptonessie.org/
2. Biham, E., Shamir, A.: Differential Cryptanalysis of DES-like Cryptosystems. Journal of Cryptology 4(1), 3–72 (1991)
3. Blaum, M., Roth, R.M.: On Lowest Density MDS Codes. IEEE Transactions on Information Theory 45(1), 46–59 (1999)
4. Brown, W.C.: Matrices over commutative Rings. Monographs and textbooks in pure and applied mathematics. Marcel Dekker, Inc. (1993)
5. Celler, F., Leedham-Green, C.R.: Calculating the Order of an Invertible Matrix. In: Finkelstein, L., Kantor, W.M. (eds.) Groups and Computation II. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 28, pp. 55–60. AMS (1997)

6. 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)
7. Daemen, J., Rijmen, V.: The Design of Rijndael: AES – The Advanced Encryption Standard. Springer (2002)
8. Guo, J., Peyrin, T., Poschmann, A., Robshaw, M.: The LED Block Cipher. In: Preneel, B., Takagi, T. (eds.) CHES 2011. LNCS, vol. 6917, pp. 326–341. Springer, Heidelberg (2011)
9. Guo, J., Peyrin, T., Poschmann, A.: The PHOTON Family of Lightweight Hash Functions. In: Rogaway, P. (ed.) CRYPTO 2011. LNCS, vol. 6841, pp. 222–239. Springer, Heidelberg (2011)
10. Kang, J., Hong, S., Lee, S., Yi, O., Park, C., Lim, J.: Practical and Provable Security Against Differential and Linear Cryptanalysis for Substitution-Permutation Networks. ETRI Journal 23(4), 158–167 (2001)
11. MacWilliams, F.J., Sloane, N.J.A.: The Theory of Error-Correcting Codes. North-Holland Publishing Company (1978)
12. Matsui, M.: Linear Cryptanalysis Method for DES Cipher. In: Helleseth, T. (ed.) EUROCRYPT 1993. LNCS, vol. 765, pp. 386–397. Springer, Heidelberg (1994)
13. Sajadieh, M., Dakhilalian, M., Mala, H., Sepehrdad, P.: Recursive Diffusion Layers for Block Ciphers and Hash Functions. In: Canteaut, A. (ed.) FSE 2012. LNCS, vol. 7549, pp. 385–401. Springer, Heidelberg (2012)
14. 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)
15. Zheng, Y., Matsumoto, T., Imai, H.: On the Construction of Block Ciphers Provably Secure and Not Relying on Any Unproved Hypotheses. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 461–480. Springer, Heidelberg (1990)

A Condition Set of $A^{(5)}_{lfsr}$

$A^{(5)}_{lfsr} = [1, L^2, L^{-1}, L^{-1}, L^2]$ with $d = 5$ is perfect, if the following 21 matrices are non-singular.

$$\begin{gathered}
L, L+1, L^2+L+1, L^3+L+1, L^3+L^2+1, L^4+L^3+1, \\
L^4+L^3+L^2+L+1, L^5+L^2+1, L^5+L^3+1, L^5+L^3+L^2+L+1, \\
L^5+L^4+L^3+L+1, L^5+L^4+L^2+L+1, L^6+L^3+1, L^6+L^5+1, \\
L^6+L^5+L^4+L+1, L^6+L^4+L^3+L+1, \\
L^7+L^3+1, L^7+L^5+L^2+L+1, L^8+L^7+L^2+L+1, \\
L^{10}+L^6+L^5+L+1, L^{10}+L^9+L^8+L^6+L^4+L^2+1.
\end{gathered}$$

Private Stream Search at Almost the Same Communication Cost as a Regular Search

Matthieu Finiasz[1] and Kannan Ramchandran[2,⋆]

[1] CryptoExperts
[2] UC Berkeley

Abstract. Private Stream Search allows keyword-based search queries to be performed on streaming data (or on a database) without revealing any information about the keywords being searched. Using homomorphic encryption, Ostrovsky and Skeith proposed a solution to this problem in 2005. However, their solution requires the server to send an answer of size $O(mS \log m)$ bits when m documents of S bits match the query, while a regular (non-private) query only requires mS bits. Following this work, some improved schemes have been proposed with the aim of keeping the reply from the server *linear* in mS. In this work we propose two new *communication optimal* constructions: both allow communication linear in mS, but they also offer an expansion factor (compared to a non-private query) asymptotically equal to 1 when m and S increase. More precisely, our first scheme requires $m(S + O(\log t))$ bits (where t is the size of the database) and our second scheme $m(S + C)$ where C is a constant depending only on the chosen computational security level.

Keywords: privacy, keyword search, Reed-Solomon codes, LDPC codes.

1 Introduction

Internet search engines are able to gather a lot of information on users from the content of the search queries they make. Most users don't really care about this, but more and more users would prefer the servers not to learn *anything* from their query. This is the goal of Private Stream Search (PSS) algorithms: being able to perform a keyword-based search query on a server, without disclosing any information about the keywords in the query. However, even if the number of users having concerns about privacy is growing, the number of these users that are willing to trade efficiency for privacy is probably much smaller. Compared to standard non-private search, PSS should not have a much higher latency (response time) or bandwidth usage (response size) or be less reliable (miss some matching documents). This is the main focus of this article.

The first PSS algorithm was introduced by Ostrovsky and Skeith [12,13] in 2005 and makes clever use of homomorphic encryption to hide the content of the query while still allowing the server to perform some computations on it.

⋆ This research was funded by the NSF grant CCF-0964018.

This scheme requires the use of a public dictionary of possible keywords and is restricted to OR queries. These restrictions are not the main focus of our work, but as we discuss later, the use of a *fully homomorphic encryption* scheme could remove these restrictions from both the original Ostrovsky-Skeith construction and the new schemes we present here. Following Ostrovsky and Skeith's work, some improvements have been proposed independently by Bethencourt, Song and Waters [1,2] and by Danezis and Díaz [4,5]. These improvements have the same structure as the original scheme and are focused mainly on improving the size of the response from the server (one of the main issues in the original proposal) and the reliability of the scheme. However, they are suboptimal in some aspects.

The main contribution of this paper is the proposal of two new PSS algorithms combining the ideas of the Ostrovsky and Skeith construction with results from coding theory, thus allowing for state of the art results that improve significantly on current PSS schemes. Our first scheme uses Reed-Solomon codes [15] and allows for a zero-error guarantee, while offering optimal communication rates. It can however be computationally heavy at the server. Our second scheme is based on irregular LDPC codes [6,10] and is asymptotically optimal, thus interesting when a large number of documents (in practice, a few hundreds) match the query. We also propose an offline-online scheme, with a higher offline computational cost, but which allows the online step to be *as efficient* as a standard non-private search: the response suffers no latency and the communication overhead remains minimal.

This article is organized in 3 sections. Section 2 contains a description of the original Ostrovsky-Skeith PSS construction and of the Bethencourt *et al.* and Danezis and Díaz improvements. Then, in Section 3, we present our two new constructions and give some analysis of their performances. Finally, in Section 4, we detail some further improvements that apply to our schemes, but also to the previous PSS schemes.

2 Previous Constructions

2.1 Paillier's Encryption Scheme

All known private stream search constructions require the use of homomorphic encryption and the original Ostrovsky-Skeith construction relies on Paillier's cryptosystem [14]. The new schemes we present in Section 3 also rely on this scheme. Of course, any other homomorphic encryption scheme could be used instead, with only minor modifications to the PSS schemes.

Paillier's cryptosystem is a public key cryptosystem: in our PSS applications, a user and a server will communicate, and all encryptions will be done with respect to the user's key. We shall denote by $\mathcal{E} : \mathbb{Z}_N \mapsto \mathbb{Z}_{N^2}$ Paillier's encryption function and by $\mathcal{D} : \mathbb{Z}_{N^2} \mapsto \mathbb{Z}_N$ the associated decryption function. As the encryption is randomized, the same message can have different associated ciphertexts, decrypting to the same value. We thus introduce the notation $y \equiv y'$ which is equivalent to $\mathcal{D}(y) = \mathcal{D}(y')$: y and y' are encryptions of the same message. With these notations, the homomorphic property can be expressed as:

$$\mathcal{E}(x_1) \times \mathcal{E}(x_2) \equiv \mathcal{E}(x_1 + x_2) \quad \text{and} \quad \mathcal{E}(x)^c \equiv \mathcal{E}(c \times x) \text{ for any } c \in \mathbb{Z}_N.$$

Additionally, Paillier's cryptosystem is semantically secure, so the server cannot distinguish between $\mathcal{E}(0)$ and $\mathcal{E}(1)$. For simplicity, we will consider that N is 1024 bits long, but it should of course be chosen according to the required security level.

The Damgård-Jurik Extension. Semantic security requires a randomized encryption, which necessarily induces a message expansion: the ciphertext is larger than the associated message. Paillier's cryptosystem has a constant expansion factor of 2, for small messages of $\log N$ bits, but also for longer messages spanning several encrypted blocks.

To improve this, Damgård and Jurik [3] proposed a variant of Paillier's homomorphic encryption scheme which works very similarly but takes a message in $\mathbb{Z}_{N^s}$ (for any value of s) and outputs a ciphertext in $\mathbb{Z}_{N^{s+1}}$. For a message of $s \log N$ bits, the expansion factor is only $\frac{s+1}{s}$, which tends to 1 when the message size increases. However, the price to pay for this smaller expansion rate is a factor $O(s^2)$ on the cost of encryption/decryption.

Using the Damgård-Jurik encryption scheme with a modulus N of 1024 bits, the communication overhead is 1024 bits whatever the message size. More generally, this overhead is a constant C depending only on the required security level.

2.2 The Ostrovsky-Skeith Construction

A private stream search algorithm works in three steps: first the user builds a query and sends it to the server, then the server executes the query which outputs a result that it sends back to the user, finally the user extracts the queried documents from the result he received. Here is the description of these three algorithms for the original PSS scheme from Ostrovsky and Skeith [12,13].

Query Construction. Let $\Omega = \{w_1, w_2, \ldots, w_{|\Omega|}\}$ be the dictionary of possible keywords and $K \subseteq \Omega$ be the set of keywords the user wants to query. The query is $Q = \{q_1, q_2, \ldots, q_{|\Omega|}\}$, where $q_j = \mathcal{E}(\mathbf{1}_{w_j \in K})$ and $\mathbf{1}$ denotes the indicator function. The user thus sends an encrypted bit for each element in the dictionary: this bit is 1 if the keyword is part of the user's search, 0 otherwise. As each encryption is independently randomized and due to the semantic security of Paillier's cryptosystem, the server cannot tell which of these encrypted bits are 1 and 0.

As part of the query, the user also sends m, the expected number of matching documents, and γ, a reliability parameter (a larger γ gives a better probability of recovering all matching documents).

Query Execution. Upon receiving the query (Q, m, γ), the server first creates a buffer B of size $\ell = \gamma m$ and initializes each of its positions to the value $1 \equiv \mathcal{E}(0)$.

Let us assume the database contains t documents. Then, for each document $f_i \in \mathbb{Z}_N$ in the database, the server computes the set $W_i \subseteq \Omega$ of keywords in the dictionary that match document f_i. It then computes $F_i = \left(\prod_{w_j \in W_i} q_j\right)^{f_i} \equiv \prod_{w_j \in W_i} \mathcal{E}(\mathbf{1}_{w_j \in K})^{f_i}$. Thanks to the homomorphic property of $\mathcal{E}$, we also have: $F_i \equiv \mathcal{E}\big(f_i \sum_{w_j \in W_i} \mathbf{1}_{w_j \in K}\big)$. Denoting $c_i = \sum_{w_j \in W_i} \mathbf{1}_{w_j \in K}$ the number of keywords of K that match f_i, we then have $F_i = \mathcal{E}(c_i f_i)$. The server then selects γ random positions $b_i = \{b_{i,1}, \ldots, b_{i,\gamma}\} \subset [1, m\gamma]$ of the buffer B and updates each of these γ positions by multiplying its current value by F_i.

After processing all the documents in the database, the j-th buffer position will be equal to $B_j = \prod_i F_i^{\mathbf{1}_{j \in b_i}} \equiv \mathcal{E}\big(\sum_i \mathbf{1}_{j \in b_i} c_i f_i\big)$, that is, the encryption of a linear combination of documents in the database. This linear combination is sparse if only few documents match the query K, meaning most of the c_i are equal to 0. The server then sends the buffer B back to the user.

In practice, everything happens as if the server had a random binary matrix H of size $\gamma m \times t$ with γ ones in each column and it was computing $B \equiv \mathcal{E}(H \times (c_i f_i)_{i \in [1,t]})$.

Document Extraction. When receiving the encrypted buffer B, the user starts by decrypting each buffer position to get $\mathcal{D}(B_j) = \sum_i \mathbf{1}_{j \in b_i} c_i f_i$. He then scans the γm decrypted buffer positions for what we call *singletons*: buffer positions that contain only one file, that is, positions such that $\mathbf{1}_{j \in b_i} c_i = 0$ for all but one value of i. The user discards all buffer positions that are not singletons and extracts the value f_i of one document from each singleton.

Asymptotic Cost. The encrypted buffer that is sent back by the server to the user has size γm. In order for the user to recover the m matching documents with a high probability of success, γ must be of the order of $O(\log m)$. If documents are S bits long, using Paillier's cryptosystem, encrypted buffer positions should be $2S$ bits long and the answer is thus of order $2mSO(\log m)$. It can be reduced to $m(S + 1024)O(\log m)$ using the Damgård-Jurik extension (with a 1024-bits modulus N). The expansion factor is only logarithmic in m, but some improvements are needed to keep the buffer size *linear* in the number of matching documents.

2.3 The Danezis-Díaz Improvement

In [4,5], Danezis and Díaz propose to modify slightly Ostrovsky and Skeith's construction, allowing them to dramatically decrease the size of the buffer B. Their algorithm works exactly like the Ostrovsky-Skeith scheme, but with the following modifications:

– the query Q is the same, but the user chooses a target buffer size ℓ (typically $\ell = 2m$),

- before processing the query, the server embeds the index i of each document inside f_i,
- when processing the query, instead of adding the document to γ random buffer positions, the server now uses a public hash function/pseudo-random number generator h, seeded by the index i, to deterministically choose a set of d positions where the document f_i will be added. The integer d is a parameter of the system. In terms of matrices, H is such that its i-th column is $H_i = h(i)$, a binary vector of Hamming weight d.

The main change that Danezis and Díaz bring is a new document extraction algorithm. As in the Ostrovsky-Skeith construction, the user starts by looking for singletons and extracts the value of some documents from these singletons. However, now, each document also contains its index i, and from i, the user can generate $H_i = h(i)$ and know exactly where this document was added in the buffer. He can thus completely remove any document he recovers from the buffer, thereby uncovering new singletons. This simple iterative decoding algorithm allows for a much smaller buffer than in the original scheme, while maintaining a decoding cost linear in the buffer size.

This algorithm was only empirically tested by Danezis and Díaz, but their experimental results show that a buffer size of $\ell = 2m$ is sufficient to recover a high percentage of the m matching documents. However, they do not give any formal analysis.

2.4 The Bethencourt-Song-Waters Construction

In [1,2], Bethencourt, Song and Waters propose a completely different angle to improve on the Ostrovsky-Skeith scheme. Their approach is useful when documents are long (more than $\log N$ bits), which will be the case in many applications. In the previous schemes, the expansion ratio between the number of matching documents and the buffer size was independent of the document size. Meaning that even for large documents, the expansion would be $\log m$ for the Ostrovsky-Skeith scheme and 2 for the Danezis-Díaz variant.

Bethencourt *et al.* propose to use a similar scheme, but with 3 different buffers. The first buffer will contain the value of the documents (and will thus scale with the size of the documents), and the two other buffers are *independent* of the document size and are used to send the values of the c_i and the indexes i of the matching documents.

With this technique, Bethencourt *et al.* are able to achieve an asymptotic expansion rate of 1 (using the Damgård-Jurik cryptosystem), but their technique suffers a few drawbacks:

- recovering the indexes i of matching documents relies on Bloom filters and requires the reply to be of size $O(m \log \frac{t}{m})$ to have a good probability of success. This dependence in the size t of the database is not desirable and, as we will see, not necessary.
- recovering the document values requires to solve a linear system of size $m \times m$, which can be quite expensive compared to the rest of the document extraction process. Other algorithms have a linear cost in the buffer size.

3 Two New PSS Constructions

Building on the original Ostrovsky-Skeith scheme, we propose two new *communication optimal* constructions. The key idea is to consider that the job of the server is simply to compute a sparse encrypted vector $\mathcal{E}(c_i f_i)$ and then compress it *in the encrypted domain.* Thanks to the homomorphic property of Paillier's cryptosystem it is possible to compute the syndrome of this vector with respect to the parity check matrix H of an error correcting code. Using two optimal code constructions (Reed-Solomon codes [15] and irregular LDPC codes [6,10]) we obtain our two new PSS schemes. Of course, the idea of using a linear function to compress a sparse vector is not new, but homomorphic encryption allows to do this in the encrypted domain too. This is also what previous PSS schemes were doing, without explicitly stating it, and using sub-optimal linear compression.

3.1 A Zero-Error Construction Using Reed-Solomon Codes

Apart from the communication overhead, one drawback of the existing PSS constructions is that they have a non-zero probability of failure. This is true even for very large expansion rates. In particular, when the number m of matching documents is small, the failure probability of all previous schemes becomes higher for a given expansion rate. We thus propose a deterministic algorithm that will guarantee that the document extraction will never fail if the number of matching documents is known. This algorithm uses Reed-Solomon codes and exploits their MDS (maximum distance separable) property in the following way:

– Reed-Solomon codes can correct up to m errors using $2m$ syndromes,
– they can also correct m erasures (errors at a known position) using only m syndromes.

Description of the Construction. A direct application of this would consist in replacing the matrix H in the PSS algorithm by the parity check matrix of a Reed-Solomon code over $\mathbb{Z}_N$. Then, recovering the value of any m documents would be equivalent to correcting m errors with the code and would only require $\ell = 2m$ buffer positions. However, it is possible to do better than this. Indeed, this straightforward application allows documents of $\log N$ bits, but also allows a database of up to N elements. In practice the database is much smaller than N (which for security reasons will be at least 2^{1024}), and using a Reed-Solomon code on $\mathbb{Z}_N$ is a waste. The server can encode the values of the c_i (and their positions) as errors in a smaller Reed-Solomon code, and then encode the documents as erasures, which can be efficiently recovered once the c_i are known. Here is how the algorithm works.

Query Construction. This step is the same as for the other algorithms (described in Section 2.2). Along with the query Q, the user sends an expected number of results m.

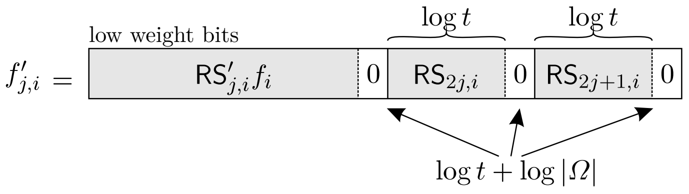


Fig. 1. Embedding of two Reed-Solomon codes and of f_i inside an element of $\mathbb{Z}_N$. $|\Omega|$ and t are the dictionary and the database sizes: the zero-paddings allow to avoid overflows when computing linear combinations of $f'_{j,i}$.

Query Execution. As in the previous constructions, for each document f_i, the server computes the encryption $\mathcal{E}(c_i)$ of the number of queried keywords matching f_i. Then, instead of simply computing $\mathcal{E}(c_i)^{f_i}$, the server will embed several values in an integer $f'_{j,i} \in \mathbb{Z}_N$ as shown in Fig. 1. This requires two different Reed-Solomon codes RS and RS'. The code RS will be used to recover the c_i and is defined on $\mathbb{Z}_{p_t}$ where p_t is the smallest prime greater than the database size t (it should also be greater than the dictionary size $|\Omega|$) and the coefficients of its parity check matrix are thus defined as $\mathsf{RS}_{j,i} = i^j \mod p_t$. Similarly, RS' will be used to recover the values of f_i and is defined over $\mathbb{Z}_{p_f}$ where p_f is the largest prime that can fit in the remaining bits of one plaintext (Paillier or Damgård-Jurik). We have $\mathsf{RS}'_{j,i} = i^j \mod p_f$. Thus, for each of the m positions of buffer B, the server multiplies B_j by $\mathcal{E}(c_i)^{f'_{j,i}}$. Once the whole database has been processed, $B_j \equiv \mathcal{E}\big(\sum_{i=1}^{t} c_i f'_{j,i}\big)$.

Document Extraction. As usual, the user starts by decrypting the buffer B. He then splits each plaintext he obtains in 3 parts: the first part corresponding to $\sum_i c_i \mathsf{RS}'_{j,i} f_i$, the second to $\sum_i c_i \mathsf{RS}_{2j,i}$ and the third to $\sum_i c_i \mathsf{RS}_{2j+1,i}$. These linear combinations have been computed in the encrypted domain by the server, with no reduction modulo p_t or p_f: this is why some zero-padding is required (see Fig. 1) to avoid overflows. The user thus starts by reducing the last two elements modulo p_t for each $j \in [1, m]$, and gets $2m$ syndrome positions in RS of the sparse vector (c_i). This is enough to recover the values and positions of the m non-zero c_i elements using the Reed-Solomon error correcting algorithm.

Then, the user reduces the first part of each plaintext modulo p_f to obtain m syndrome positions in RS' of the sparse vector $(c_i f_i)$. As the non-zero c_i elements are known, the positions of the non-zero $c_i f_i$ are also known, and the user has to solve an erasure problem. The m syndrome positions are enough to recover the values of $c_i f_i$, and thus also of f_i.

Computational Cost. Compared to the Ostrovsky-Skeith construction, the use of Reed-Solomon codes has an heavy impact on the server side computations. As all the lines of a Reed-Solomon parity check matrix are different, the server has to compute a modular exponentiation $\mathcal{E}(c_i)^{f'_{j,i}}$ for each coefficient in the matrix, that is mt exponentiations instead of t in the other algorithms.

However, on the user side, the computational cost remains similar and will be dominated by the buffer decryption step. Reed-Solomon decoding costs $O(m^2)$ multiplications in $\mathbb{Z}_{p_t}$ and $\mathbb{Z}_{p_f}$, which can be upper bounded by $O(m^2(\log N)^2)$. Decryption costs $O(m(\log N)^3)$, which will dominate as long as m is smaller than a few thousands. The document extraction process is thus no longer linear in m, but the quadratic component is negligible in practice.

Communication Cost. With this scheme, a buffer of size m is enough to recover m matching documents, which is optimal. However, part of each buffer position is reserved for the recovery of the c_i and for zero-padding. For each document, an overhead of $5 \log t + 3 \log |\Omega|$ bits has to be transmitted. The $3 \log t + 3 \log |\Omega|$ padding bits[1] are wasted bits that are due to the structure of the Paillier encryption scheme: using a different homomorphic encryption scheme could improve this. The remaining $2 \log t$ bits are however necessary to get a deterministic zero-error algorithm: the user has to solve an error correction problem, meaning he will have to learn both the value and *the position* of the errors, leading to an overhead of $O(\log t)$ bits per document. Overall, for m matching documents of S bits, this scheme requires $2m(S + O(\log t))$ bits using the original Paillier cryptosystem or $m(S + 1024 + O(\log t))$ using the Damgård-Jurik extension. Asymptotically, when S tends to infinity, this corresponds to an expansion ratio (compared to a non-private search) of 2 using Paillier and 1 using Damgård-Jurik.

For non-asymptotic parameters, suppose we take a database of 1 000 billion documents, a dictionary with 1 million keywords and a query for 5 keywords returning 200 results, and we use the smallest possible zero-paddings of $\log m + \log |K|$ bits. The reply from the server would be $200 \times 2\,048 = 409\,600$ bits long, and would consist of 204 800 bits of randomness (added by Paillier's encryption), $11 \times 3 \times 200 = 6\,600$ bits of padding, $40 \times 2 \times 200 = 16\,000$ bits to recover the c_i, and the remaining 182 200 bits to contain the information. This gives an expansion ratio of only 2.25 for these small parameters, and this ratio will improve when the document size increases.

Note that this analysis is valid only if the user knows in advance the number m of matches to expect. This can be the case in some scenarios, but not in a typical keyword search. In this case, the user will query for m positions and will receive a syndrome of size m: if less than m documents match the query he will be able to extract all of them with probability 1, but if more than m documents match the query he will not be able to decode and will not get *any* document. This gives another way of looking at the zero-error property: the user knows when he misses something, whereas in the Ostrovsky-Skeith scheme, the user will usually be able to extract at least a few documents, but has no information whether he got all the documents or not. With our Reed-Solomon scheme, it is possible to imagine an interactive protocol[2] where the user can

[1] In practice, this can be reduced to $3 \log m + 3 \log |K|$ bits, but having a dependency on m is not really convenient and revealing $|K|$ to the server leaks some information.

[2] Special care has to be taken when designing such an interactive protocol, so as not lose too much privacy by disclosing the actual number of matches to the server.

query for additional syndrome positions when the decoding fails, thus allowing him to choose a small m at first and still be sure to get all the documents in the end.

3.2 An Asymptotically Optimal Construction Using Irregular LDPC Codes

Description of the Construction. In order to improve the asymptotic communication cost and remove any dependency on the database size t, it is necessary to use a randomized scheme (thus with a non-zero probability of failure): in that case, it is well known that (irregular) LDPC codes can offer much better performance than Reed-Solomon codes. However, the error correction problem also has to be transformed into an erasure correction problem. This is possible by combining the following ideas (which is similar to what Danezis and Díaz proposed):

– instead of using a fix LDPC matrix, generate it from the documents f_i themselves,
– use a decoding algorithm similar to the erasure correction algorithm proposed by Luby and Mitzenmacher for verification codes [8].

Query Construction. This step is the same as for the Ostrovsky-Skeith construction. Instead of m and γ, the user sends the desired buffer length ℓ to the server.

Query Execution. The server first initializes a buffer B of size ℓ to $\mathcal{E}(0)$ in every position. Then, for each document f_i it proceeds as follows:

– compute f_i' from f_i as shown in Fig. 2, adding a padding (with a single 1 between some zeros) and a small non-linear checksum of f_i itself (a cryptographic hash of 64 bits can be used),
– compute $\mathcal{E}(c_i f_i')$ exactly as in the original scheme,
– using a pseudo-random number generator seeded by f_i, generate[3] an integer d following a given distribution (the best choice for this distribution is discussed at the end of this section) and generate a uniformly random binary column vector H_i of length ℓ and Hamming weight d,
– for every non-zero position in H_i, multiply the corresponding position in buffer B by $\mathcal{E}(c_i f_i')$.

In the end, $B = \mathcal{E}(H \times (c_i f_i'))$ contains the encrypted syndrome of the sparse $(c_i f_i')$ vector with respect to the parity check matrix H of an irregular LDPC code.

[3] Note that Danezis and Díaz use the document index i to generate the column H_i, which requires to number documents. Using the document itself as the seed makes application to streaming data more natural.

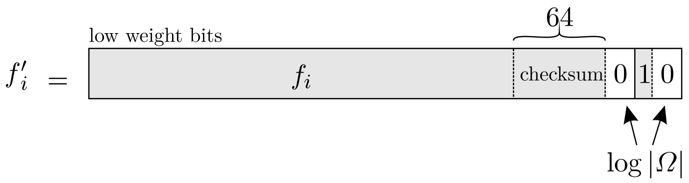


Fig. 2. Embedding of f_i, a padding and a checksum of f_i inside an element of $\mathbb{Z}_N$. $|\Omega|$ is the dictionary size (a bound on c_i) and the paddings avoid overflows when multiplying f_i' by c_i. In practice, this padding can be reduced to $\log|K|$ bits, so just a couple of bits for queries with a few keywords.

Document Extraction. When receiving buffer B, the user starts by decrypting it and looks for singletons. The detection of singletons is made easy by the structure of f_i' (see Fig. 2). The isolated 1 allows to read the value of c_i directly and divide $c_i f_i'$ by it: if it is indeed a singleton, the checksum will be valid and the value of f_i can be recovered, otherwise the checksum will not be valid (with high probability).

With every singleton the user gets the value of one document f_i and can now regenerate (using the same PRNG as the server) the corresponding column H_i. Knowing H_i (and c_i) it is possible to remove document f_i from the other syndrome positions where it has been added, thus uncovering new singletons, which in turn can reveal new documents f_i. This gives an iterative algorithm which we analyse asymptotically here.

Choosing an Optimal Column Weight Distribution. In order to analyze the decoding algorithm, the matrix H can be transformed into a bipartite graph. On the left of the graph there are m information nodes (the non-zero $(c_i f_i')$ elements) and on the right there are ℓ parity nodes (the decrypted syndromes). These nodes are connected by edges: each 1 in H is an edge in the graph, linking an information node and a parity node. For each edge in the graph, its *left degree* is the number of edges connected to its information node and its *right degree* the number of edges connected to its parity node. Then the decoding algorithm consists in repeating the following steps:

– select all edges with right degree 1 (edges connected to singletons),
– remove these edges from the graph as well as the associated left and right nodes,
– remove all other edges that were connected to the left nodes (no other edges were connected to the right nodes).

Decoding is successful if, at the end, all the edges have been removed from the graph.

Studying the probability of success of this algorithm for given parameters m and ℓ is difficult, however, as proven in [9], if the left and right degree distribution of edges remains constant and m and ℓ tend to infinity, the asymptotic proportion of edges removed at each step can be computed.

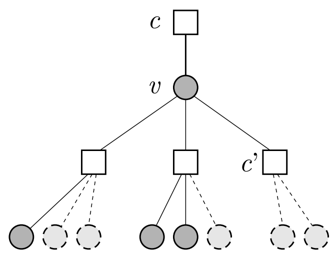


Fig. 3. Local view of the bipartite graph as a tree. The dashed lines correspond to nodes/edges removes at the end of step j. The edge between v and c will be removed at step $j+1$ as one son c' of v is a singleton (it has no sons remaining at the end of step j).

Let $\lambda(x) = \sum_i \lambda_i x^{i-1}$ and $\rho(x) = \sum_i \rho_i x^{i-1}$, where λ_i (resp. ρ_i) denote the probability that an edge of the graph has left (resp. right) degree i. Also, let b_j denote the proportion of edges of the graph that are still present after step j of the algorithm. Then $b_0 = 1$ (all the edges are present before the algorithm starts) and:

$$b_{j+1} = \lambda(1 - \rho(1 - b_j)). \tag{1}$$

We will not prove this formula here (please refer to [9,11] for the complete proof), but the intuition is the following. During the decoding process, the neighborhood around the edge (v, c) between a message node v and a parity node c, can be seen as a tree (see Fig. 3). The decoding process then consists in letting information flow from the leaves of the tree to its root: the (v, c) edge will be removed from the graph at step $j+1$ if the value of v can be determined at step $j+1$ and it can thus send its value to c. As can be seen on Fig. 3, the value of the message v can be determined if it has a son c' that is a singleton at the end of step j of the algorithm, meaning one of the sons of v is a leaf. The probability this happens can easily be computed if the distributions λ and ρ are known. The parity node c' is connected to k edges with probability ρ_k and it is a singleton at the end of step j if its $k-1$ sons in the Fig. 3 tree have been removed: this happens with probability $P_{singleton} = \sum_k \rho_k (1-b_j)^{k-1} = \rho(1-b_j)$.

Similarly, node v has k' neighbors (and thus $k'-1$ sons) with probability $\lambda_{k'}$, so one of them will be a singleton with probability $1 - \sum_{k'} \lambda'_k (1 - P_{singleton})^{k'-1} = 1 - \lambda(1 - P_{singleton})$. The edge (v, c) is thus removed at step $j+1$ with probability $1 - \lambda(1 - \rho(1 - b_j))$, which leads to formula (1).

Asymptotically, the decoding algorithm is successful if $b_j \overset{j\to\infty}{\longrightarrow} 0$, which will be the case if:

$$\forall x \in [0,1], \quad \lambda(1 - \rho(1 - x)) \leq x.$$

Of course, for finite values of m and ℓ, the algorithm can fail even if this condition is verified. The sequence of b_j represents the expected evolution of the number of edges in the graph, but the algorithm will deviate from the average from time to time.

Table 1. Minimal expansion rates asymptotically allowing recovery of all the documents, for specific values of d, in the algorithm of Danezis and Díaz

d	2	3	4	5	6	7	8	9
minimal $\frac{\ell}{m}$	2	1.2218	1.2949	1.4249	1.5697	1.7189	1.8692	2.0192

Application to Danezis and Díaz's Algorithm. In [4], Danezis and Díaz propose to use a constant weight d for the columns of H. This means that $\lambda(x) = x^{d-1}$. The distribution ρ is a little more complicated as it is induced by λ. A row of H will have a weight following a binomial distribution: it is the sum of m random coins equal to 1 with probability $\frac{d}{\ell}$. We thus have $\rho_i = \frac{i\ell}{md}\binom{m}{i}\left(\frac{d}{\ell}\right)^i\left(1-\frac{d}{\ell}\right)^{m-i}$ and so, when m is large, $\rho(x) = \exp\left(-\frac{md}{\ell}(1-x)\right)$.

The best rate $\frac{\ell}{m}$ one can achieve for a given d is given in Table 1 and is obtained from the equation:

$$\forall x \in [0,1], \quad \left(1 - \exp(-\tfrac{md}{\ell}x)\right)^{d-1} \leq x.$$

The best asymptotic choice is $d = 3$, which does not mean that for a given m and ℓ the best probability of success is always achieved for $d = 3$. This however proves that Danezis and Díaz's algorithm with constant column weight d can indeed achieve communications linear in m. However, it can never have a good probability of success for expansion rates smaller than 1.22.

The Harmonic Distribution. Using a constant column weight makes the description of the algorithm easy and gives very good results for some parameters (see Fig. 5), but it is not optimal.

The decoding problem we are facing is very close to the problem of decoding LT-codes [7] and, for LT-codes, it was proven that the optimal distribution choice is the so-called *Robust Soliton* distribution. However, in our context, using this distribution would correspond to fixing $\rho(x)$, that is, choosing a row weight distribution. This is not possible as each instance of our decoding problem corresponds to a set of m random columns of H, meaning we only have full control on the column weight distribution $\lambda(x)$. The row distribution will always be given by $\rho(x) = \exp\left(-\frac{m\tilde{d}}{\ell}(1-x)\right)$, where $\tilde{d} = 1/\sum_i \frac{\lambda_i}{i}$ is the average column weight of H. Thus, the constraint on $\lambda(x)$ is:

$$\forall x \in [0,1], \quad \lambda\left(1 - \exp(-\tfrac{m\tilde{d}}{\ell}x)\right) \leq x. \tag{2}$$

With a change of variable $y = 1 - \exp(-\frac{m\tilde{d}}{\ell}x)$, inequality (2) becomes:

$$\forall y \in \left[0, 1 - \exp(-\tfrac{m\tilde{d}}{\ell})\right], \quad \lambda(y) \leq -\tfrac{\ell}{m\tilde{d}}\ln(1-y).$$

The optimal choice is thus the harmonic distribution, consisting in a normalized Taylor series expansion of $-\ln(1-x)$ truncated at order D. It is given by:

$$\lambda_D(x) = \frac{1}{H(D)}\sum_{i=2}^{D}\frac{1}{i-1}x^{i-1} \quad \text{with} \quad H(D) = \sum_{i=2}^{D}\frac{1}{i-1}.$$

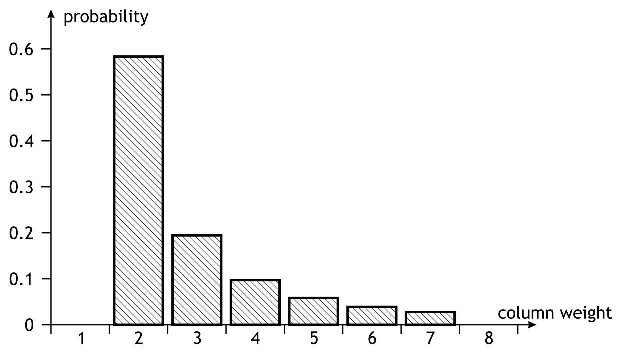


Fig. 4. Column weight distribution for the harmonic distribution of order 7

These $\lambda_D(x)$ distributions will satisfy inequality (2) if the expansion factor $\frac{\ell}{m}$ is greater than $1 + \frac{1}{D}$. Any expansion ratio $\frac{\ell}{m} = 1 + \epsilon$ can thus be chosen by the user, and using the harmonic distribution of order $\frac{1}{\epsilon}$ will asymptotically allow to recover most documents.

The Enhanced Harmonic Distribution. One problem with the harmonic distribution is the presence of columns of weight 2 (see Fig. 4). Any distribution containing columns of weight 2 will have a non-zero asymptotic probability of having identical columns[4]. Indeed, the collision probability P_{col} that 2 of the m matching columns of H are identical is $P_{\text{col}} \simeq \sum_i \binom{\frac{\tilde{d}\lambda_i}{i}m}{2} / \binom{\ell}{i}$. If the ratio $\frac{\ell}{m}$ is constant and m tends to infinity, all the terms in this sum tend to 0, except the term $i = 2$. Asymptotically, $P_{\text{col}} \overset{m\to\infty}{\longrightarrow} \left(\frac{\tilde{d}\lambda_2 m}{2\ell}\right)^2$, which is not negligible for the harmonic distribution.

This means that, using the harmonic distribution, decoding will often end with a few ($O(1)$ asymptotically) unrecovered documents. In order to deal with this issue, one solution is to combine this distribution with a constant weight distribution to obtain what we call the *enhanced harmonic distribution.* Each column of H is the concatenation of a column of length $\ell - \ell_3$ following the harmonic distribution and a column of length ℓ_3 of constant weight 3.

In practice, the proportion of collisions tends to 0 when m grows, so the length ℓ_3 can be chosen such that $\frac{\ell_3}{m}$ tends to 0 and the probability of full recovery still tends to 1. For example, $l_3 = O(\sqrt{\ell})$ would be a reasonable choice. Asymptotically, these ℓ_3 additional rows in H do not increase the expansion factor. Using this enhanced harmonic distribution with an order D harmonic distribution, the probability of recovering all m documents tends to 1 when $\frac{\ell}{m} > 1 + \frac{1}{D}$ and m tends to infinity.

[4] If two documents f_i generate the exact same column H_i, our decoding algorithm will be unable to recover any of them as no singleton can ever be obtained.

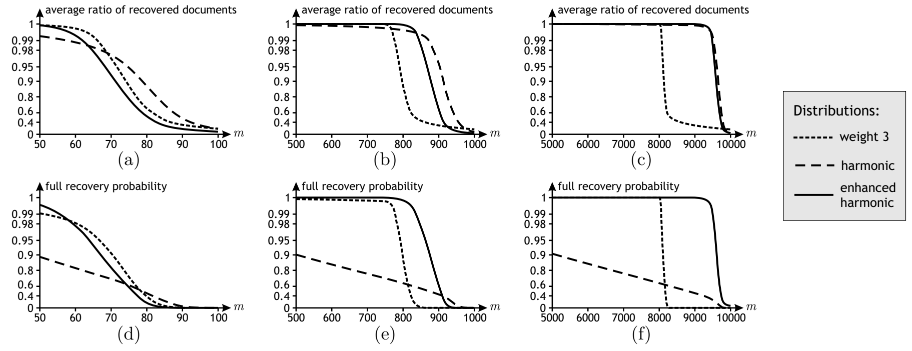


Fig. 5. Simulation results for different LDPC weight distributions. The curves represent the average ratio of recovered documents and the probability of recovering all m documents as a function of the number of matching documents m for different buffer sizes ℓ. In (a) and (d) $\ell = 100$ and $\ell_3 = 10$, in (b) and (e) $\ell = 1000$ and $\ell_3 = 35$, and in (c) and (f) $\ell = 10000$ and $\ell_3 = 100$.

Communication Cost. We compared the different distributions through simulations and the results we obtained are shown in Fig. 5. One can clearly see the limitation of the weight 3 distribution: for $m > .8\ell \simeq \frac{\ell}{1.22}$ the probability of full recovery drops. However, the enhanced harmonic distribution holds to our expectations, it behaves as expected with a probability of recovering all m matching documents close to 1 even for expansion rates smaller than 5%.

With this algorithm and the enhanced harmonic distribution, the asymptotic communication cost of private stream search with m matching documents of S bits is thus $2m(S + 64 + O(\log |\Omega|))$ using Paillier's cryptosystem, or $m(S + 1088 + O(\log |\Omega|))$ using the Damgård-Jurik extension, which corresponds to an expansion rate of 1 compared to a non-private search.

4 Further Improvements

An Offline-Online Construction. Using any of our two new schemes, it is possible to reduce the size of the reply from the server almost to the size of a non-private search result. However, the size of the query the user sends remains large. The size of a private query is linear in $|\Omega|$ whereas it is logarithmic for a non-private query. To improve this, we propose an offline-online scheme, where the linear query is sent offline and a logarithmic query is sent online.

A Single Keyword Scheme. We first focus on queries containing a single keyword. In this case, any query can be obtained as a cyclic shift of any other query. The scheme works as follows:

– offline, the user generates a query $Q_j = \{q_1, \dots, q_{|\Omega|}\}$ where $q_j = \mathcal{E}(1)$ and $q_i = \mathcal{E}(0)$ otherwise (with j picked uniformly at random), and sends it to the server,

- offline, the server computes all possible cyclic shifts of Q_j by $i \in [0, [\Omega| - 1]$ positions and executes the corresponding queries. It stores each result in a separate buffer B_i.
- online, the user wants to query the server for the j'-th keywords and sends $j' - j \mod |\Omega|$ to the server,
- online, the server sends $B_{j'-j}$ to the user and discards the other B_i,
- the user decrypts/decodes buffer $B_{j'-j}$ normally.

With this scheme, the online work on the server side is a simple table lookup and the amount of online communication is very close to the non-private case: the query is only $\log |\Omega|$ bits long, and with the PSS schemes we have presented $B_{j'-j}$ can also be small.

The offline amount of communication is still the same as for the standard scheme, but the amount of computation on the server side is multiplied by $|\Omega|$. However, as this is offline work, it can easily be outsourced to distant server farm and does not have to be run on the "online" low-latency servers. Of course, if the amount of offline work is too high, it is also possible to treat the shifted query online as in the standard scheme: the amount of work the server has to do will then be the same as in the normal scheme, but most of the communication will be done offline.

Dealing with Multiple Keywords. To maintain privacy, the offline query the user sends has to be completely random and, at the same time, it should be possible to modify it into any other query the user might later want to ask. For a single keyword, cyclic shifts work well whatever the dictionary size. However, for several keywords (say k), the user should be able to transform any random query $Q_{j_1,\dots,j_k}$ into any chosen query $Q_{j'_1,\dots,j'_k}$, by simply giving the index of a permutation.

When $k = 2$, a solution is to transform each index j into $Aj + B \mod |\Omega|$, where $A \in [1, |\Omega| - 1]$ and $B \in [0, |\Omega| - 1]$ are the "permutation index" that the user will send to the server in the online phase. If $|\Omega|$ is prime, then any pair j_1, j_2 can be transformed into any pairs j'_1, j'_2 by choosing $A = \frac{j_2 - j_1}{j'_2 - j'_1} \mod |\Omega|$ and $B = Aj_1 - j'_1 \mod |\Omega|$.

Building such families of permutations for larger values of k is not always simple, and the number of permutations in the family will always have to be at least $\binom{|\Omega|}{k}$. This means that the offline work on the server side will be $O(|\Omega|^k)$ times more than in the standard online scheme. Values of k larger than 1 or 2 are therefore not very realistic, and for these values the solutions we presented work fine.

Using Fully Homomorphic Encryption. The PSS schemes we presented do not need fully homomorphic encryption (FHE) to work: computing a syndrome only requires addition and multiplication by a scalar. However, having an efficient FHE scheme would allow AND queries, whereas the current schemes are limited to OR queries. Also, the query size could be reduced to $O(\log |\Omega|)$ encrypted bits without having to use an offline-online scheme[5]. For instance, when querying for keyword w_i, the user could decompose the index i in $\log|\Omega|$ bits and send two ciphertexts $\mathcal{E}(0)$ and $\mathcal{E}(1)$ (or $\mathcal{E}(1)$ and $\mathcal{E}(0)$) for each of these bits. The server can then regenerate the whole query Q by homomorphically multiplying the encrypted bits corresponding to each index. Continuing on this idea, it would even be possible to get rid of the dictionary by simply querying a bit string: a query of $2n$ encrypted bits would be enough to search any pattern of n bits.

One interesting aspect of this application of FHE, is that the number of homomorphic multiplications that have to be done is independent of the database size: multiplications are required to reconstruct the query, but not for the stream search itself. Even if homomorphic multiplication is very expensive, or if the chosen scheme only allows for a few multiplications, it is still possible use it for very large databases: only the dictionary size is limited.

Also, one should note that the homomorphic encryption scheme used in PSS does not have to be a public key scheme: only the user encrypts and decrypts elements (the server simply has to be able to initialize the buffer to $\mathcal{E}(0)$, but the user could send him this initial value). A symmetric homomorphic encryption scheme with a limit on the number of possible multiplications could thus also be of interest if it allows a small message expansion rate.

Application to Set Reconciliation. An interesting aspect of the LDPC scheme we presented is that, even though it behaves like a randomized scheme, it is fully deterministic. What this means is that if the user already knows a subset of the documents f_i that are going to match the query, he can compute a buffer B for the documents he knows in the exact same way as the server and "remove" this buffer from the reply he gets from the server.

This is the idea of set reconciliation: two users A and B know two sets S_A and S_B and want to learn the elements the other user knows with the minimal amount of communication. Classical results from set reconciliation show that the amount of communication required is only $|S_A| + |S_B| - 2|S_A \cap S_B|$.

The same optimal result can be obtained with our construction: if m documents match the query, but the user already knows m' of these documents, a buffer of size $m - m'$ is (asymptotically) enough. This way, if a user repeats the same search every day (for example, to monitor changes to a database), the amount of communication required can be reduced even further by keeping a cache of the previous search results.

5 Conclusion

We presented two new private stream search constructions allowing minimal communication from the server to the user. Using Paillier's cryptosystem, compared to a standard non-private search, the response from the server in our

[5] Not that a query of $O(\log|\Omega|)$ bits is probably not achievable in practice as current FHE schemes all require some important message expansion, but a poly-logarithmic query is possible.

Table 2. Comparison of the computational and communication complexities of the various PSS schemes. $|\Omega|$ is the dictionary size, t is the number of documents in the database, S is the size of a document in the database, and m is the number of documents matching the query. $n = \log N$ is the size of the modulus used in Paillier's encryption and $\frac{S}{n}$ is the number of plaintext blocks required to encrypt one document. Reply sizes are given assuming the use of the original Paillier encryption: using the Damgård-Jurik scheme can gain a factor 2 in the reply sizes of all schemes.

	query size (in bits)	server complexity	reply size (in bits)	user complexity
Non-private search	$\|K\| \log \|\Omega\|$	t	Sm	Sm
Ostrovsky-Skeith	$2n\|\Omega\|$	tn^3	$2Sm \log m$	$\frac{S}{n} m \log mn^3$
Bethencourt *et al.*	$2n\|\Omega\|$	tn^3	$2Sm + 2nm + 2nm \log m$	$m(\frac{S}{n} + \log m)n^3 + m^{2.376}$
	$2n\|\Omega\|$	tn^3	$2Sm + 2nm + 2nm \log \frac{t}{m}$	$m(\frac{S}{n} + \log \frac{t}{m})n^3 + m^{2.376} + t \log \frac{t}{m}$
Danezis-Díaz	$2n\|\Omega\|$	tn^3	$2.44(S + \log t)m$	$1.22 \frac{S}{n} mn^3$
Our RS scheme	$2n\|\Omega\|$	mtn^3	$2(S + 5 \log t + 3 \log \|\Omega\|)m$	$\frac{S}{n} mn^3 + m^2 n^2$
Our LDPC scheme	$2n\|\Omega\|$	tn^3	$2(S + 2 \log \|\Omega\|)m$	$\frac{S}{n} mn^3$

schemes only suffers a factor 2 expansion. Using the Damgård-Jurik extension, this expansion factor can be made as close to one as required when the document size tends to infinity. Also, our Reed-Solomon based construction allows zero-error rate, meaning the user is guaranteed to get the documents he expects (or, if he chose m too small, he will know he missed some documents). This comes at a cost on the server side, but can be a very important feature for some applications.

The improvements we presented here do not solve all the problems of private stream search: the computational cost on the server side is still much more than for a regular search, and the size of the query the user has to send is also a problem. Still, our LDPC scheme offers better performances than all the previous private stream search constructions, without any additional computations. Table 2 gives a comparison of the asymptotic costs of the different PSS schemes mentioned here.

References

1. Bethencourt, J., Song, D.X., Waters, B.: New constructions and practical applications for private stream searching (extended abstract). In: 2006 IEEE Symposium on Security and Privacy, pp. 132–139. IEEE Computer Society (2006)
2. Bethencourt, J., Song, D.X., Waters, B.: New techniques for private stream searching. ACM Trans. Inf. Syst. Secur. 12(3) (2009)
3. Damgård, I., Jurik, M.: A Generalisation, a Simplification and Some Applications of Paillier's Probabilistic Public-key System. In: Kim, K.-C. (ed.) PKC 2001. LNCS, vol. 1992, pp. 119–136. Springer, Heidelberg (2001)

4. Danezis, G., Díaz, C.: Improving the decoding efficiency of private search. In: Anonymous Communication and its Applications. Dagstuhl Seminar Proceedings, vol. 05411. IBFI, Schloss Dagstuhl, Germany (2006)
5. Danezis, G., Diaz, C.: Space-Efficient Private Search with Applications to Rateless Codes. In: Dietrich, S., Dhamija, R. (eds.) FC 2007 and USEC 2007. LNCS, vol. 4886, pp. 148–162. Springer, Heidelberg (2007)
6. Gallager, R.G.: Low-density parity-check codes. M.I.T. Press, Cambridge (1963)
7. Luby, M.G.: LT codes. In: FOCS, pp. 271–280. IEEE (2002)
8. Luby, M.G., Mitzenmacher, M.: Verification codes. In: Proc. Allerton Conf. on Communication, Control, and Computing (2002)
9. Luby, M.G., Mitzenmacher, M., Shokrollahi, M.A.: Analysis of random processes via and-or tree evaluation. In: SODA, pp. 364–373 (1998)
10. Luby, M.G., Mitzenmacher, M., Shokrollahi, M.A., Spielman, D.A.: Analysis of low density codes and improved designs using irregular graphs. In: Proceedings of the 30th Annual ACM Symposium on Theory of Computing, STOC 1998, pp. 249–258. ACM (1998)
11. Luby, M.G., Mitzenmacher, M., Shokrollahi, M.A., Spielman, D.A.: Efficient erasure correcting codes. IEEE Transactions on Information Theory 47(2), 569–584 (2001)
12. Ostrovsky, R., Skeith, W.E.: Private Searching on Streaming Data. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 223–240. Springer, Heidelberg (2005)
13. Ostrovsky, R., Skeith, W.E.: Private searching on streaming data. Journal of Cryptology 20(4), 397–430 (2007)
14. Paillier, P.: Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In: Stern, J. (ed.) EUROCRYPT 1999. LNCS, vol. 1592, pp. 223–238. Springer, Heidelberg (1999)
15. Reed, I.S., Solomon, G.: Polynomial codes over certain finite fields. Journal of the SIAM 8(2), 300–304 (1960)

An Optimal Key Enumeration Algorithm and Its Application to Side-Channel Attacks

Nicolas Veyrat-Charvillon, Benoît Gérard, Mathieu Renauld, and François-Xavier Standaert

UCL Crypto Group, Université catholique de Louvain, Place du Levant 3, B-1348, Louvain-la-Neuve, Belgium

Abstract. Methods for enumerating cryptographic keys based on partial information obtained on key bytes are important tools in cryptanalysis. This paper discusses two contributions related to the practical application and algorithmic improvement of such tools. On the one hand, we observe that the evaluation of leaking devices is generally based on distinguishers with very limited computational cost, such as Kocher's Differential Power Analysis. By contrast, classical cryptanalysis usually considers large computational costs (e.g. beyond 2^{80} for present ciphers). Trying to bridge this gap, we show that allowing side-channel adversaries some computing power has major consequences for the security of leaking devices. For this purpose, we first propose a Bayesian extension of non-profiled side-channel attacks that allows us to rate key candidates according to their respective probabilities. Then we provide a new deterministic algorithm that allows us to optimally enumerate key candidates from any number of (possibly redundant) lists of any size, given that the subkey information is provided as probabilities, at the cost of limited (practically tractable) memory requirements. Finally, we investigate the impact of key enumeration taking advantage of this Bayesian formulation, and quantify the resulting reduction in the data complexity of various side-channel attacks.

1 Introduction

Side-channel attacks represent an important threat to the security of cryptographic hardware products. As a consequence, evaluating the information leakage of microelectronic circuits has become an important part in the certification of secure devices. Most of the tools/attacks that have been published in this purpose are based on a so-called "divide-and-conquer" approach. That is, in a first "divide" part, the evaluator/adversary recovers information about different parts of the master key, usually denoted as subkeys (as a typical example, the target can be the 16 AES key bytes). Next, a "conquer" part aims to combine the information gathered in an efficient way, in order to recover the full master key.

Research over the last ten years has been intensive in the optimization of the divide part of attacks. Kocher et al.'s Differential Power Analysis (DPA) [14] and Brier et al.'s Correlation Power Analysis (CPA) [6] are notorious examples.

One limitation of such approaches is their somewhat heuristic nature, as they essentially rank the subkeys according to scores that do not have a probabilistic meaning. As demonstrated by Junod in the context of linear cryptanalysis, such heuristic key ranking procedures may be suboptimal compared to Bayesian key recoveries [12]. The template attacks introduced by Chari et al. in 2002 typically aim to get rid of this limitation [7]. By carefully profiling a probabilistic model for the physical leakages, such attacks offer a direct path towards Bayesian subkey testing procedures. Template attacks are optimal from an information theoretic point of view, which makes them a prime tool for the worst-case security evaluation of leaking devices [32]. However, they also correspond to strong adversarial assumptions that may not be met in practice. Namely, actual adversaries are not always able to profile an accurate enleakage model, either because of a lack of knowledge of the target devices or because of physical variability [26]. As a consequence, attacks profiling an "on-the-fly" leakage model such as the stochastic approach, introduced by Schindler et al. [27] and discussed by Doget et pal. [9], are an important complement to a worst-case security analysis.

By contrast, only little attention has been paid to the conquer part in side-channel analysis. That is, in most cases the attacks are considered successful if all the subkeys are recovered with high confidence, which generally implies an extremely small time complexity for the offline computations. This situation is typically exemplified by initiatives such as the DPA contest [22], where the success rate in recovering a master key is directly obtained as the success rates for the concatenated 16 subkeys ranked first. In fact, the most noticeable exceptions attempting to better exploit computational power in physical attacks are based on advanced techniques, e.g. exploiting the detection of collisions [4,15,28,29], or taking advantage of algebraic cryptanalysis [5,20,24,25], of which the practical relevance remains an open question (because of stronger assumptions). But as again suggested by previous works in statistical cryptanalysis, optimal key ranking procedures would be a more direct approach in order to better trade data and time complexities in "standard" side-channel attacks.

In this paper, we improve the divide and the conquer parts of side-channel attacks, with two main contributions. Regarding the divide part, we start with the observation that non-profiled side-channel attacks are usually limited by their heuristic use of scores when ranking subkey candidates. As a consequence, we propose a method for non-profiled attacks that allows deriving probability mass functions for the subkey hypotheses. This tool can be viewed as a natural extension of the stochastic approach, but is also applicable to DPA and CPA. More generally, expressing the information obtained through non-profiled side-channel attacks with probabilities allows us to connect them better with template attacks, where the scores are already expressed as subkey probabilities.

Second, we provide the first comprehensive investigation of the conquer part of side-channel attacks. For this purpose, we start from the motivation that testing several billions of key candidates on a modern computer is not far-fetched: being able to recover a master key after such a computation is indeed a security breach. Next, we observe that two main solutions for testing key candidates from partial information on the subkeys exist in the literature. On the one hand, Meier and Staffelbach proposed a sampling algorithm in 1991. However, it turns out that in our side-channel attack context, such a probabilistic search leads to significant overheads in terms of number of keys to test. On the other hand, Pan, van Woudenberg, den Hartog and Witteman described a deterministic key enumeration algorithm at SAC 2010. But large memory requirements prevent the application of this second solution when the number of keys to enumerate increases. For example in [21], the authors were limited to the enumeration of 2^{16} candidates. As none of these tools is perfectly suited to our side-channel attack context, we propose a new deterministic algorithm for key enumeration, that is time and memory efficient, and allows the optimal enumeration of full keys by decreasing order of probabilities. It takes advantage of the probability mass functions of subkeys made available through our first contribution. The new algorithm can be viewed as an improvement of the SAC 2010 one, in which we reduce the memory complexity of the enumeration thanks to a recursive decomposition of the problem. Interestingly, and as previously observed in other cryptanalysis settings, this improvement of the key ranking strategy also has a significant impact on the data complexity of side-channel key recoveries.

Summarizing, this work brings an interesting complement to the evaluation framework in [32]. It allows stating standard side-channel attacks as a data complexity vs. time complexity tradeoff. On the theoretical side, the proposed key enumeration algorithm leads to a proper estimation of security metrics such as high-order success rates or guessing entropy, for block cipher master keys (this estimation was previously limited to subkeys or small orders). In practice, experimental results also exhibit that considering adversaries with a reasonable computing power leads to significant improvements of standard side-channel attacks. These gains are particularly interesting if we compare them with the ones obtained by only working on the statistics in the divide part of side-channel attacks [31]. Hence, we believe that the tools introduced in this paper have an important impact for the security evaluations of leaking devices, e.g. for certification laboratories. In this respect, it is worth noticing the gap between the computational complexities usually considered in the evaluation of side-channel attacks and the ones considered for evaluating security against mathematical attacks [16,23]. We also note that the tools introduced in this paper are generic and have potential impact in other cryptanalytic settings (e.g. based on faults [3], or statistical [2,18]), although standard side-channel attacks are a very natural environment for using them. Finally, in order to stimulate authors to consider the computational aspect of side-channel attacks, we provide an optimized implementation of the enumeration algorithm available in [1].

2 Background

The “standard” side-channel attacks considered in this work use a divide-and-conquer approach in which the side-channel subkey recovery phase focuses on one specific operation at a time [17]. In block ciphers like the AES, this operation is usually one 8-bit S-box in the first encryption round. We denote with p and k the byte of the plaintext and the byte of the key (i.e. the subkey) that are used in the attack, with $x = p \oplus k$ the input value of the S-box, and with $y = \mathsf{S}(x)$ the corresponding output value. The goal of a side-channel subkey recovery phase is to identify the best (and hopefully correct) subkey candidate $\hat{k}$ from the set of all possible key hypotheses $\mathcal{K}$, using q measured encryptions. For each target S-box the adversary collects a data set of pairs $\{(p_i, l_i)\}_{1 \leq i \leq q}$[1], with p_i the i^{th} plaintext byte involved in the target S-box computation, and l_i the corresponding leakage value. For simplicity, and because it has little impact on our following discussions, we assume unidimensional leakages. In addition, we assume leakage samples composed of a deterministic and a random part, with the deterministic part depending only on the S-box output (i.e. we use the EIS assumption introduced in [27]). The leakage samples can consequently be written as $l_i = \mathsf{L}(y_i) = \mathsf{f}(y_i) + n$, with n a Gaussian distributed noise. In general, side-channel attacks can be classified as profiled and non-profiled attacks, depending on whether the adversary can perform a training phase or not.

Profiled attacks, like template attacks [7], take advantage of their profiling phase to characterize the leaking device with a probabilistic model. This allows the adversary to rank the subkey hypotheses k according to their actual probabilities: $\hat{k} = \arg\max_k \Pr[k|\{(p_i, l_i)\}]$. These probabilities can then directly be used to build a Probability Mass Function (PMF): $f_K(k) = \Pr[k|\{(p_i, l_i)\}]$, with K the discrete random variable corresponding to the unknown subkey byte. This PMF will be needed to perform the key enumeration in Section 4. By contrast, in the case of non-profiled attacks (e.g. DPA [14] or CPA [6]), the best subkey hypothesis is not chosen based on probabilities, but on the value produced by a statistical distinguisher (namely, a difference-of-means test for Kocher's DPA and Pearson correlation coefficient for CPA). For these non-profiled attacks, there is thus no straightforward way to produce the PMF we need to enumerate the master keys. That is, the distinguisher outputs a ranking of the subkey candidates, which has no probabilistic meaning.

In order to apply our key enumeration algorithm, we need a way to extract probabilities from a non-profiled attack. For this purpose, we will use a natural extension of the non-profiled version of Schindler's stochastic approach [27]. Hence, we first recall how the non-profiled stochastic attack works [9]. A stochastic model $\theta(y)$ is a leakage model used to approximate the leakage function: $\mathsf{L}(y) \simeq \theta(y)$, where $\theta(y)$ is built from a linear basis $\mathsf{g}(y) = \{\mathsf{g}_0(y), ..., \mathsf{g}_{B-1}(y)\}$ chosen by the adversary (usually $\mathsf{g}_i(y)$ are polynomials in the bits of y). Evaluating $\theta(y)$ boils down to estimating the coefficients α_i such that the vector $\theta(y) = \sum_j \alpha_j \mathsf{g}_j(y)$ is a least-square approximation of the measured leakages l_i. The idea of a non-profiled stochastic attack is to build $|\mathcal{K}|$ stochastic models $\theta_k(y)$ by considering the data set $\{(p_i, l_i)\}$ under the assumption that the correct key hypothesis is k. These stochastic models are then used as a distinguisher: for a correct key hypothesis (and a relevant basis), the error between the predicted values and the actual leakage values should have a smaller standard deviation

[1] In order to lighten the notations, we omit the index $1 \leq i \leq q$ after the data sets.

than for a wrong key hypothesis. The pseudo-code of the attack is given in Algorithm 1. In general, an interesting feature of such attacks is that they allow trading robustness for precision in the models, by adapting the basis $\mathsf{g}(y)$. That is, a simpler model with less parameters is more robust, but a more complex model can potentially more accurately approximate the real leakage function.

Algorithm 1. Non-profiled stochastic attack

```
1: Acquire $\{(p_i, l_i)\}_{1 \le i \le q}$.
2: Choose a basis $\mathsf{g}(y)$.
3: for $k \in \mathcal{K}$ do
4:     Compute the S-box output hypotheses $y_{i,k} = \mathsf{S}(p_i \oplus k)$.
5:     Use the basis $\mathsf{g}(y)$, the data set and the subkey hypothesis $k$
       in order to build a stochastic model $\theta_k$.
6:     Compute the error vector $e_k$: $e_{i,k} = l_i - \theta_k(y_{i,k})$.
7:     Evaluate the precision of the model: $\sigma_k =$ standard deviation$(e_k)$.
8: end for
9: Choose $\hat{k} = \arg\min_k \sigma_k$.
```

3 Bayesian Extension of Non-profiled SCAs

As the straightforward application of a stochastic attack does not produce PMFs, we propose in this section to perform an additional Bayesian step after building the stochastic models. We show that this Bayesian model comparison produces probabilities, and that the criterion of maximizing the likelihood of the subkey is equivalent to minimizing the error vector standard deviation, meaning that this extension indeed ranks the subkeys in the same order as the standard non-profiled stochastic attack. As a bonus, we observe that this extension also gives us a very natural way to combine independent leakage samples in an attack. In the Bayesian version of the non-profiled stochastic attack, we perform a Bayesian hypothesis test on subkey candidates (under the assumption that the basis used for the stochastic attack is valid). It consists in estimating the probability of the observed data set assuming that they are produced from the model θ_k (i.e. $\Pr[\{(p_i, l_i)\}|\theta_k]$). Then, we use Bayes' theorem to deduce the likelihood of the models (and thus the probabilities of the subkey hypotheses) given the data (i.e. $\Pr[\theta_k|\{(p_i, l_i)\}]$), as described by the pseudo-code of Algorithm 2. A detailled derivation of relevant probabilities is given in Appendix A.

Algorithm 2. Bayesian non-profiled stochastic attack

```
1 to 8: Same as Algorithm 1.
9: Perform a Bayesian model comparison: evaluate for each subkey hypothesis the
likelihood $\Pr[\theta_k|\{(p_i, l_i)\}]$ using Bayes' theorem.
10: Choose $\hat{k} = \arg\max_k \Pr[\theta_k|\{(p_i, l_i)\}]$.
```

4 A New Algorithm for Combining Subkeys

Following the evaluation framework in [32], different metrics can be used to analyze the security of an implementation against side-channel attacks. Of particular interest in this work are the so-called "security metrics" (namely, the success rate and guessing entropy), of which the goal is to estimate the efficiency of a given distinguisher in exploiting the leakage to recover keys. Intuitively, a success rate of order o corresponds to the probability that the correct key is rated among the o first candidates provided by the distinguisher. The guessing entropy corresponds to the average number of keys to test before reaching the correct one. As suggested in [21], one can also consider a guessing entropy of order o, in order to capture the fact that in practice, only a maximum number of o keys can be tested by the evaluators. Empirical comparisons of distinguishers using such metrics have been performed in [31], but were limited to subkey recoveries (i.e. key bytes, typically). In the following, we consequently tackle the (most practically relevant) problem of how to efficiently estimate these metrics for master keys. In the extreme case (i.e. success rate of order 1), the solution is straightforward, as e.g. illustrated by the DPA contest [22]. We consider the general case of large lists and large orders, to carefully address the problem of the "conquer" part in a side-channel attack. The problem of extracting the rank of a correct key is equivalent to the problem of enumerating keys stated below.

Key-Enumeration Problem. The attacker obtains PMFs corresponding to d independent subkeys, each PMF taking n different values. The problem is to enumerate complete keys from the most probable to the least probable one.

In the following, we qualify an enumeration algorithm as *optimal* if it outputs key candidates in *nonincreasing* order of posterior probability. The term optimal refers to the fact that this order minimizes the expected number of key trials. Any non-optimal algorithm incurs an overhead in terms of trials during the key recovery phase. Besides, the more subjective term *efficient* relates to the computational and memory-related costs of the algorithm. For example, the naive algorithm for solving the enumeration problem generates the list of all possible key candidates, computes the corresponding likelihood values (by multiplying subkey probabilities) and sorts them accordingly. While optimal in the previously described sense, it can only be used with key candidates lists of limited size, and is therefore very inefficient. In the remainder of this section, we propose an algorithm that optimally solves the enumeration problem, and allows time and memory efficient key-enumeration, even when the number and size of subkey lists makes the naive approach untractable.

4.1 An Optimal and Efficient Key-Enumeration Algorithm

The key enumeration problem can be more readily understood as a geometric problem. We first consider the simpler bi-dimensional case (i.e. 2 subkeys). The key space can be identified with a compartimentalized square of length 1. The enumeration process is illustrated in Figure 1. The 4 columns (resp. rows) correspond to the four possible values of the first (resp. second) subkey, sorted by nonincreasing order of probability. Width and height correspond to the probability of the corresponding subkey. Let us denote by $k_i^{(j)}$ the j^{th} likeliest value for the i^{th} subkey. Then, the intersection of row j_1 and column j_2 is a rectangle corresponding to the key $(k_1^{(j_1)}, k_2^{(j_2)})$ with probability equal to the area of the rectangle. Using this geometric view, an optimal key enumeration algorithm outputs compartments by nonincreasing order of area. A solution to this problem is given in Algorithm 3 and corresponds to the following steps:

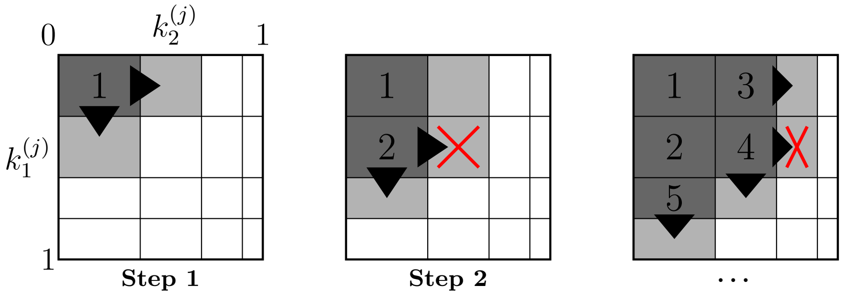


Fig. 1. Geometric representation of the proposed algorithm

Step 1. The most likely key is $(k_1^{(1)}, k_2^{(1)})$. Hence, we output this one first (represented in dark gray). At this point, the only possible next key candidates are the successors $(k_1^{(2)}, k_2^{(1)})$ and $(k_1^{(1)}, k_2^{(2)})$, shown in light gray. We denote by $\mathcal{F}$ this set of potential next candidates (where $\mathcal{F}$ is standing for frontier).

Step 2. Any new candidate has to belong to the frontier set. We extract the most likely candidate from this set and output it. It corresponds to rectangle 2 in our example. $\mathcal{F}$ is updated by inserting the potential successors of this candidate.

Next steps. Step 2 is repeated until the correct key is output, or if the size of the frontier set $\mathcal{F}$ exceeds the available memory space.

Note that in step 2, $(k_1^{(2)}, k_2^{(1)})$ is a more likely candidate than $(k_1^{(2)}, k_2^{(2)})$ by construction. Hence, $(k_1^{(2)}, k_2^{(2)})$ should not be inserted into $\mathcal{F}$ (this is represented on the figure by red crosses). There is a simple rule for handling such cases, which allows minimizing the memory requirements of our algorithm:

Rule 1. The set $\mathcal{F}$ may contain at most one element in each column and row.

Operations on the frontier set can be performed efficiently if candidate keys are stored in an ordered structure. Indeed, these operations simply consist in inserting new elements or finding the most likely element in the set and removing it. Using heaps, these manipulations are logarithmic in the size of the set. The test of Rule 1 can be implemented using arrays of Boolean values.

Algorithm 3. Optimal key-enumeration.

```
$\mathcal{F} \longleftarrow \{(k_1^{(1)}, k_2^{(1)})\}$;
while $\mathcal{F} \neq \emptyset$ do
    $(k_1^{(i)}, k_2^{(j)}) \longleftarrow$ most likely candidate in $\mathcal{F}$;
    Output $(k_1^{(i)}, k_2^{(j)})$;
    $\mathcal{F} \longleftarrow \mathcal{F} \setminus \{(k_1^{(i)}, k_2^{(j)})\}$;
    if $i+1 \leq \#k_1$ and no candidate in row $i+1$ then
        $\mathcal{F} \longleftarrow \mathcal{F} \cup \{(k_1^{(i+1)}, k_2^{(j)})\}$;
    end if
    if $j+1 \leq \#k_2$ and no candidate in column $j+1$ then
        $\mathcal{F} \longleftarrow \mathcal{F} \cup \{(k_1^{(i)}, k_2^{(j+1)})\}$;
    end if
end while
```

Generalization to Multiple Lists. In practice, one often has to merge together more than two lists of subkeys. Straightforward extensions of our algorithm to higher dimensions lead to either suboptimal or slow rules for frontier set reduction. On the one hand, the direct transposition of Rule 1 will minimize memory, but implies adjacency tests in multiple dimensions, leading to unacceptable reductions of the enumeration speed. On the other hand, simplifying the rule in order to maintain a good enumeration speed implies the storage of many non-necessary candidates in the frontier set, which rapidly leads to unsustainable memory requirements. As a result, and in order to obtain good results for more than two lists, we apply a recursive decomposition of the problem.

For this purpose, we only use the algorithm for merging two lists, and its outputs are combined to form larger subkey lists which are in turn merged together. This way, merging n lists is done by merging two lists $n-1$ times. The order of merging is such that lists merged together are of similar sizes. Taking the example of the AES, we notice that enumerating 128-bit keys is done by merging two lists of size 2^{64}. Such lists cannot be generated or stored efficiently. Fortunately, we can instead generate these lists only as far as required by the key enumeration. Whenever a new subkey is inserted in the candidate set, we get it from the enumeration algorithm applied to the lower level (e.g. 64-bit subkeys are obtained by merging two 2^{32} element lists), and so on. This ensures that the storage and enumeration effort are minimized. The process is illustrated in Figure 2. Enumerating 16-byte keys consists in enumerating subkeys taken from the two 2^{64}-element lists $k_{0,\ldots,7}^{(j)}$ and $k_{8,\ldots,15}^{(j)}$, which in turn are built using four 2^{32}-element lists $k_{0,\ldots,3}^{(j)}$, $k_{4,\ldots,7}^{(j)}$, etc. This process is repeated until we reach the original 2^8-element subkey distributions. The recursive decomposition combined with our *lazy evaluation* technique keep computations and memory requirements to a minimum and allow us to enumerate a large number of key candidates.

The investigations in this paper have connections with previous works in statistical cryptanalysis. We provide a detailled review in Appendix B.

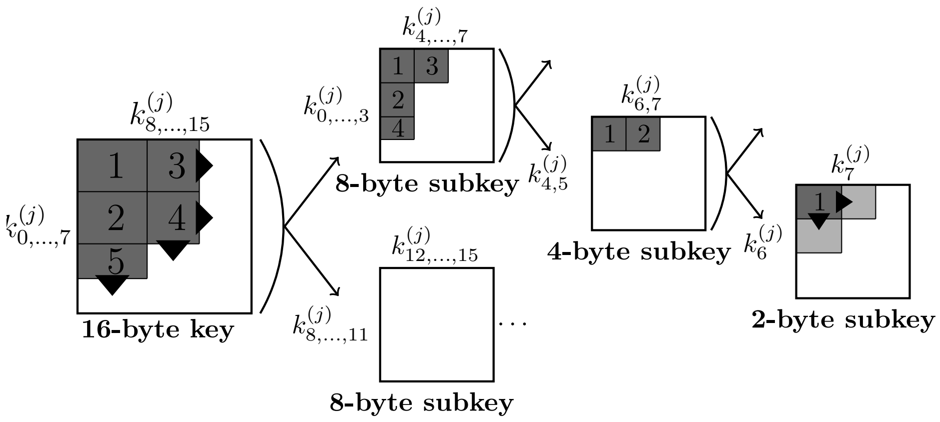


Fig. 2. Recursive enumeration from multiple lists of key candidates

5 Experiments

In order to validate our approach, we led several experiments. The cipher under investigation was the AES, with key size of 128 bits. Our side-channel attacks targeted the output of the S-boxes in the first round, resulting in 16 independent subkey probability mass functions. We used the same assumptions as in Section 2 and considered simulated leakages, following a Hamming weight leakage model on the S-box output, with an independent additive Gaussian noise, i.e. $\mathsf{f}(y) = \mathrm{HW}(y) + \mathcal{N}(0, 4^2)$. Note that the type of experiments performed (i.e. analyzing the impact of key enumeration) is essentially independent of both the cipher and experimental setup. We carried out both template attacks with perfect profiling (since the leakage function is known) and non-profiled Bayesian stochastic attacks assuming a linear basis made of the S-box output bits. The enumeration was performed using our open-source optimized implementation [1].

5.1 Comparing Optimal and Probabilistic Key-Enumerations

Table 1 gives some performance results for key enumeration obtained on our setup (Intel core i7 920 running a 64-bit Ubuntu 11.04 distribution). These comparative results show that both the sampling algorithm and ours can output key candidates at essentially the same speed. The results for the enumeration algorithm described in [21] are also given. As expected, they exhibit larger memory requirements, which bounds the number of key candidates that can be enumerated and increases time. In practice, this method is limited to 2^{20} candidates. By contrast, our algorithm allows enumerating 2^{32} keys using less than 1GB.

Next, as mentioned in Section 4, Algorithm 3 performs key trials in the best possible order, therefore minimizing the enumeration effort at the cost of a growing amount of memory space. By contrast, the probability-driven algorithm may miss some key candidates and output some more than once. In order to illustrate

Table 1. Practical comparison of key-enumeration algorithms (time, min-max memory)

#trials	2^{16}	2^{20}	2^{24}	2^{28}	2^{32}	2^{33}	2^{35}	2^{37}
Sampling	0.04s	0.31s	10.1s	160s	2560s	X	X	X
[21]	0.96s 118MB	18.1s 7.7GB	X	X	X	X	X	X
Ours	0.01s 100KB 500KB	0.25s 2MB 3MB	5s 3.9MB 12MB	100s 30MB 80MB	1700s 140MB 530MB	4058s 190MB 540MB	5.4h 560MB 1.1GB+2.3HDD	28.2h 0.9GB+1.7HDD 1.1GB +6.1HDD

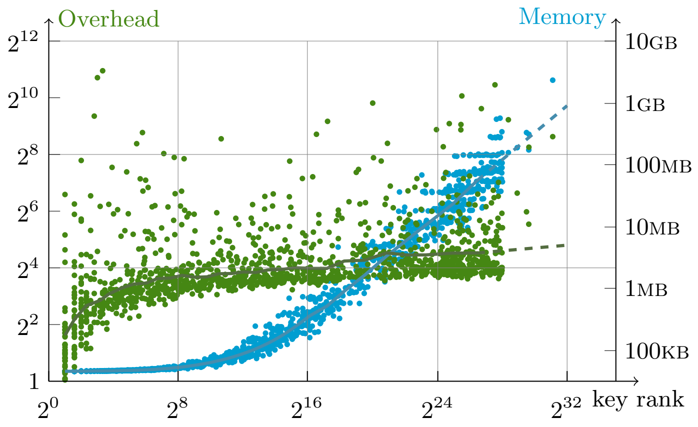


Fig. 3. Overhead of the probability-driven method in function of the key rank (green), and memory requirements of the deterministic enumeration (light blue)

these differences we led the following experiment. A large number of side-channel attacks followed by a key recovery were performed, and we measured the key rank (which is also the number of trials for the optimal algorithm), the expected number of trials for the probability-driven algorithm and the memory used during optimal enumeration. Figure 3 illustrates the expected overhead of the probability-driven method over the deterministic one in terms of key trials (green dots, left scale) and the memory cost of the enumeration algorithm (blue dots, right scale). We observe that the probability-driven algorithm requires more key trials on average to complete an attack. The overhead increases consistently, and the median of the expected ratio value (green curve) appears to tend towards a linear relation on the log-log scale. An approximated power law gives 16 for 2^{16}, 21 for 2^{32}, an extrapolated 36 for 2^{40}. In some cases, we also observe overheads very far from the median value (well over 1000), even when the correct key is ranked among the 4 first ones. On top of this *expected* number of trials, we have to consider that, since the probabilistic method follows a geometric law, the number of key trials will have a very large variance (approximately the square of expected value). This makes the probabilistic method both more costly and less reliable than our deterministic algorithm. Besides, the memory space requirement of the enumeration method also appears to follow a power law. Enumerating up to 2^{32} requires only 1GB of memory. Extrapolations predict a cost of 70GB for 2^{40} keys.

5.2 Application of Key-Enumeration to Side-Channel Attacks

Figure 4 illustrates the success rate of different orders for a template attack, in function of the number of traces measured. The alternated light and gray zones correspond to the evolution of the success rate each time the number of tested keys is multiplied by 16. The rightmost dark gray curve is obtained by only testing the first key candidate, the first light gray curve by testing 2^4 keys, then 2^8, ... We again considered the optimal and the probabilistic algorithms, for different number of enumerated key candidates. The optimal enumeration was led up to 2^{32} candidates, and the probabilistic one up to 2^{28}.

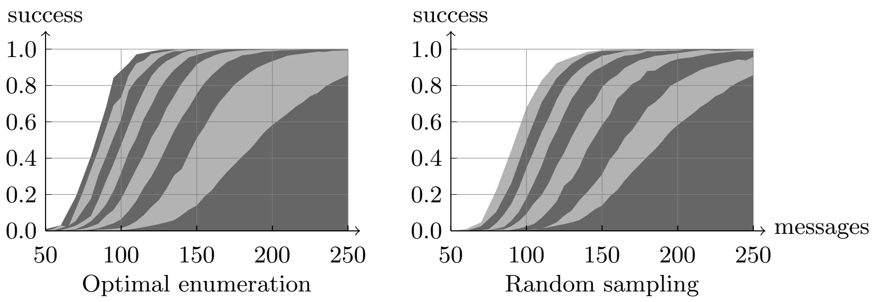


Fig. 4. Success rate of template attacks. Left: enumeration, right: sampling.

As expected, allowing more key candidates to be tested can dramatically increase the efficiency of a key recovery. For 120 messages measured, the best key candidate is the correct one about 2% of the time, while there is a 91% chance for the correct key to be found among the first 2^{24} candidates with the optimal enumeration algorithm (or an 84% chance with the probability driven method). In other words, increasing the number of key trials significantly improves the success rate, thereby providing a tradeoff between the data and time complexities of the attacks. As in the previous subsection, we also observe that the optimal enumeration algorithm leads to higher success rates compared to the random sampling for a given number of key trials, at the cost of additional memory requirements.

Figure 5 provides an orthogonal view of the problem: for a given number of traces, one can increase the key success rate by enumerating more key candidates. The figure shows the cumulative probability function (CDF) of key recoveries for an attack with a fixed number of traces, in function of the number of key trials. The PMFs used for this experiment are output by two template attacks. The first attack (left) targets only 2 key bytes, the second (right) targets all 16 key

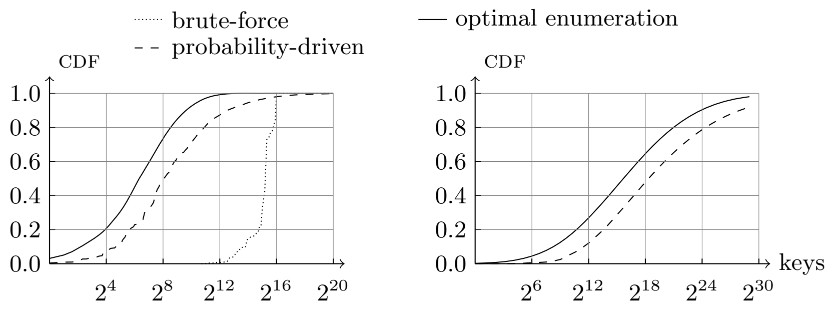


Fig. 5. Enumeration success rates. Left: 2 S-boxes, right: 16 S-boxes.

bytes. As expected, the cumulative probability starts from 0 and reaches 1 once a sufficient number of keys have been tested. Also, the brute force testing is only possible in the left case (i.e. when we can enumerate the full list).

In general, the side-channel information allows obtaining high success rates with a limited number of key trials (here, up to 2^{30}). More importantly, the figure again confirms the interest of the deterministic algorithm in terms of "number of keys to test". Reaching a similar success rate with the probabilistic algorithm requires between 2^2 and 2^4 more tests, depending on the success rates.

6 Conclusion

This paper complements standard side-channel cryptanalysis by investigating the improvements obtained by adversaries with non-negligible computational power. We first proposed an extension of non-profiled stochastic attacks that outputs probability mass functions, providing us with the likelihood values of subkey candidates. Next, we proposed a new and deterministic key enumeration algorithm, in order to take advantage of these likelihood values. Experiments show that this order-optimal enumeration algorithm is more efficient than a sampling-based algorithm from Eurocrypt 1991. In particular, the probabilistic algorithm suffers from its underlying geometric law, that implies an increasingly large overhead over the deterministic method (in terms of key trials), as the number of keys to enumerate increases. The deterministic method additionally allows removing the possibility of worst cases, which makes it a particularly suitable solution for side-channel security evaluations. Finally, our proposal significantly reduces the memory requirements of a deterministic algorithm from SAC 2010, making it the best practical solution for enumeration of up to 2^{40} keys.

As a result, the solutions in this paper allow us to properly trade side-channel measurements for offline computations. They create a bridge between classical DPA and brute-force key recovery, where information extracted through side-channels is used to improve an exhaustive search. Hence, an interesting research problem is to compare computationally-enhanced DPA attacks with other types of more computational side-channel attacks, e.g. based on the detection of collisions. The application of enumeration in statistical cryptanalysis is another possible direction for further investigations. Note finally that the complete key recoveries we considered in this work can possibly be performed in a ciphertext-only context. That is, the adversary can evaluate a key candidate by partially decrypting the ciphertext and computing the probability of previous-round leakages, which will only be non-negligible when decrypting with the correct key.

Acknowledgements. This work has been funded by the B-CCENTRE with the financial support from the Prevention of and Fight against Crime Programme of the European Union, by the the ERC project 280141 (acronym CRASH), by the Walloon regions MIPSs and SCEPTIC projects. F.-X. Standaert is a Research Associate of the Belgian Fund for Scientific Research (FNRS-F.R.S).

References

1. http://perso.uclouvain.be/fstandae/source_codes/enumeration/
2. Biham, E., Shamir, A.: Differential Cryptanalysis of DES-like Cryptosystems. In: Menezes, A., Vanstone, S.A. (eds.) CRYPTO 1990. LNCS, vol. 537, pp. 2–21. Springer, Heidelberg (1991)
3. Biham, E., Shamir, A.: Differential Fault Analysis of Secret Key Cryptosystems. In: Kaliski Jr., B.S. (ed.) CRYPTO 1997. LNCS, vol. 1294, pp. 513–525. Springer, Heidelberg (1997)
4. Bogdanov, A.: Improved Side-Channel Collision Attacks on AES. In: Adams, C., Miri, A., Wiener, M. (eds.) SAC 2007. LNCS, vol. 4876, pp. 84–95. Springer, Heidelberg (2007)
5. Bogdanov, A., Kizhvatov, I., Pyshkin, A.: Algebraic Methods in Side-Channel Collision Attacks and Practical Collision Detection. In: Chowdhury, D.R., Rijmen, V., Das, A. (eds.) INDOCRYPT 2008. LNCS, vol. 5365, pp. 251–265. Springer, Heidelberg (2008)
6. Brier, E., Clavier, C., Olivier, F.: Correlation power analysis with a leakage model. In: Joye, Quisquater (eds.) [11], pp. 16–29
7. Chari, S., Rao, J.R., Rohatgi, P.: Template Attacks. In: Kaliski Jr., B.S., Koç, Ç.K., Paar, C. (eds.) CHES 2002. LNCS, vol. 2523, pp. 13–28. Springer, Heidelberg (2003)
8. Dichtl, M.: A new method of black box power analysis and a fast algorithm for optimal key search. J. Cryptographic Engineering 1(4), 255–264 (2011)
9. Doget, J., Prouff, E., Rivain, M., Standaert, F.-X.: Univariate side channel attacks and leakage modeling. J. Cryptographic Engineering 1(2), 123–144 (2011)
10. Johansson, T. (ed.): FSE 2003. LNCS, vol. 2887. Springer, Heidelberg (2003)
11. Joye, M., Quisquater, J.-J. (eds.): CHES 2004. LNCS, vol. 3156. Springer, Heidelberg (2004)
12. Junod, P.: On the Optimality of Linear, Differential, and Sequential Distinguishers. In: Biham, E. (ed.) EUROCRYPT 2003. LNCS, vol. 2656, pp. 17–32. Springer, Heidelberg (2003)
13. Junod, P., Vaudenay, S.: Optimal key ranking procedures in a statistical cryptanalysis. In: Johansson (ed.) [10], pp. 235–246
14. Kocher, P.C., Jaffe, J., Jun, B.: Differential Power Analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999)

15. Ledig, H., Muller, F., Valette, F.: Enhancing collision attacks. In: Joye, Quisquater (eds.) [11], pp. 176–190
16. Lenstra, A.K., Verheul, E.R.: Selecting cryptographic key sizes. J. Cryptology 14(4), 255–293 (2001)
17. Mangard, S., Oswald, E., Standaert, F.-X.: One for all – all for one: unifying standard differential power analysis attacks. IET Information Security 5(2), 100–110 (2011)
18. Matsui, M.: Linear Cryptanalysis Method for DES Cipher. In: Helleseth, T. (ed.) EUROCRYPT 1993. LNCS, vol. 765, pp. 386–397. Springer, Heidelberg (1994)
19. Meier, W., Staffelbach, O.: Analysis of Pseudo Random Sequences Generated by Cellular Automata. In: Davies, D.W. (ed.) EUROCRYPT 1991. LNCS, vol. 547, pp. 186–199. Springer, Heidelberg (1991)
20. Oren, Y., Kirschbaum, M., Popp, T., Wool, A.: Algebraic Side-Channel Analysis in the Presence of Errors. In: Mangard, S., Standaert, F.-X. (eds.) CHES 2010. LNCS, vol. 6225, pp. 428–442. Springer, Heidelberg (2010)
21. Pan, J., van Woudenberg, J.G.J., den Hartog, J.I., Witteman, M.F.: Improving DPA by Peak Distribution Analysis. In: Biryukov, A., Gong, G., Stinson, D.R. (eds.) SAC 2010. LNCS, vol. 6544, pp. 241–261. Springer, Heidelberg (2011)
22. T. ParisTech. DPA contest v2, `http://www.dpacontest.org/v2/index.php`
23. C. K. L. Recommendation, `http://www.keylength.com/`
24. Renauld, M., Standaert, F.-X.: Algebraic Side-Channel Attacks. In: Bao, F., Yung, M., Lin, D., Jing, J. (eds.) Inscrypt 2009. LNCS, vol. 6151, pp. 393–410. Springer, Heidelberg (2010)
25. Renauld, M., Standaert, F.-X., Veyrat-Charvillon, N.: Algebraic Side-Channel Attacks on the AES: Why Time also Matters in DPA. In: Clavier, C., Gaj, K. (eds.) CHES 2009. LNCS, vol. 5747, pp. 97–111. Springer, Heidelberg (2009)
26. Renauld, M., Standaert, F.-X., Veyrat-Charvillon, N., Kamel, D., Flandre, D.: A Formal Study of Power Variability Issues and Side-Channel Attacks for Nanoscale Devices. In: Paterson, K.G. (ed.) EUROCRYPT 2011. LNCS, vol. 6632, pp. 109–128. Springer, Heidelberg (2011)
27. 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, pp. 30–46. Springer, Heidelberg (2005)
28. Schramm, K., Leander, G., Felke, P., Paar, C.: A collision-attack on AES: Combining side channel- and differential-attack. In: Joye, Quisquater (eds.) [11], pp. 163–175
29. Schramm, K., Wollinger, T.J., Paar, C.: A new class of collision attacks and its application to DES. In: Johansson (ed.) [10], pp. 206–222
30. Seshadri, N., Sundberg, C.-E.W.: List viterbi decoding algorithms with applications. IEEE Transactions on Communications 42(2/3/4), 313–323 (1994)
31. Standaert, F.-X., Gierlichs, B., Verbauwhede, I.: Partition *vs.* Comparison Side-Channel Distinguishers: An Empirical Evaluation of Statistical Tests for Univariate Side-Channel Attacks against Two Unprotected CMOS Devices. In: Lee, P.J., Cheon, J.H. (eds.) ICISC 2008. LNCS, vol. 5461, pp. 253–267. Springer, Heidelberg (2009)
32. Standaert, F.-X., Malkin, T.G., Yung, M.: A Unified Framework for the Analysis of Side-Channel Key Recovery Attacks. In: Joux, A. (ed.) EUROCRYPT 2009. LNCS, vol. 5479, pp. 443–461. Springer, Heidelberg (2009)

A Bayesian Extension of Non-profiled SCAs

We now show how to compute these probabilities, starting with the probability of observing the data set $\{(p_i, l_i)\}$ assuming it is produced by the model θ_k, using the subkey hypothesis k. This probability is computed by multiplying the probabilities of each individual event (p_i, l_i) of the data set:

$$\begin{aligned}\Pr[\{(p_i, l_i)\}|\theta_k] &= \Pr[\{(p_i, l_i)\}|\theta_k, K = k],\\ &= \prod_{i=1}^{q} \mathcal{N}(l_i, \theta_k(\mathsf{S}(p_i \oplus k)), \sigma_k),\end{aligned}$$

where $\mathcal{N}(x, \mu, \sigma)$ is the value of the normal distribution of mean μ and standard deviation σ evaluated at point x. If we denote the S-box output hypotheses as $y_{i,k} = \mathsf{S}(p_i \oplus k)$, the previous equation can be rewritten as:

$$\Pr[\{(p_i, l_i)\}|\theta_k] = \prod_{i=1}^{q} \frac{1}{\sqrt{2\pi}\ \sigma_k} \exp^{-\frac{1}{2\sigma_k^2}(l_i - \theta_k(y_{i,k}))^2}.$$

Since $\sigma_k^2 = \sum_{i=1}^{i=q}(l_i - \theta_k(y_{i,k}))^2/q$ (see Algorithm 1), if we use all q measurements, we can simplify the exponent and move all constants coefficients that do not depend on k in a normalization term Z, that is:

$$\Pr[\{(p_i, l_i)\}|\theta_k] = Z\sigma_k^{-q}. \tag{1}$$

Then, using Bayes' theorem, we deduce the probabilities of the subkeys from the respective likelihood values of the models θ_k given the data (in other words, we perform a Bayesian model comparison):

$$\begin{aligned}\Pr[k|\{(p_i, l_i)\}] &= \Pr[\theta_k|\{(p_i, l_i)\}],\\ &= \frac{\Pr[\{(p_i, l_i)\}|\theta_k].\Pr[\theta_k]}{\Pr[\{(p_i, l_i)\}]}, \quad \text{(Bayes' theorem)}\\ &= \frac{\Pr[\{(p_i, l_i)\}|\theta_k].\Pr[\theta_k]}{\sum_{k' \in \mathcal{K}} \Pr[\{(p_i, l_i)\}|\theta_{k'}].\Pr[\theta_{k'}]}.\end{aligned}$$

Assuming a uniform prior $\Pr[\theta_k]{=}\Pr[k]{=}\frac{1}{|\mathcal{K}|}$ and using Equation 1, we get:

$$\Pr[k|\{p_i, l_i\}] = \frac{\sigma_k^{-q}}{\sum_{k'} \sigma_{k'}^{-q}}.$$

From these probabilities, we can directly build the PMF required for key enumeration. Note that these likelihood values are not exactly the same as the ones we used in template attacks. In the last case, the characterization of a device allows exploiting a precise estimation of the leakage distributions. By contrast, in a Bayesian non-profiled stochastic attack, they depend on the basis $\mathsf{g}(y)$ chosen by the adversary. Finally, the subkey hypotheses can be ranked according to the likelihood values of the corresponding model given the data, that is:

$$\begin{aligned}\hat{k} &= \arg\max_k \sigma_k^{-q}, \\ &= \arg\min_k \sigma_k,\end{aligned}$$

i.e. providing the same ranking as for the original non-profiled stochastic attack. Besides their use for key enumeration in the next section, an appealing property of using probabilities instead of other criteria (e.g. like a correlation coefficient) is that combining independent measurement points becomes very natural. Let us suppose that our implementation leaks information at two different times: $\mathsf{L}_{t_0} = \mathsf{f}_{t_0}(x)+n_{t_0}$ and $\mathsf{L}_{t_1} = \mathsf{f}_{t_1}(y)+n_{t_1}$ (with n_{t_i} a Gaussian noise). The Bayesian writing makes it straightforward to combine their corresponding probabilities, by multiplication and normalization. Note also that the proposed attack can additionally be seen as a generalization of DPA or CPA, by simply replacing the leakage basis by a single-bit or Hamming weight model (as observed in [17]).

B Comparison with Previous Works

The investigations in this paper have strong connections with previous works in the area of statistical cryptanalysis. In particular, the problem of merging *two* lists of subkey candidates was encountered by Junod and Vaudenay [13]. The small cardinality of the lists (2^{13}) was such that the simple approach that consists in merging and sorting the lists of subkeys was tractable. Dichtl considered a similar problem of enumerating key candidates by decreasing order of probabilities, thanks to partial information obtained for each key bit individually [8]. We tackle the more general and challenging problem of exploiting any partial information on subkeys. A frequent reference for solving this problem, i.e. enumerating many keys from lists that cannot be merged, is the probabilistic algorithm that was proposed in [19]. In this work, the attacker had no access to the subkey distributions but was able to generate subkeys according to them. Hence, the solution proposed was to enumerate keys by randomly drawing subkeys according to these distributions. Implementing this algorithm is equivalent to uniformly picking up a point in the square of Figure 1 and testing the corresponding key. This does not require any memory but the most probable keys may be drawn many times, leading to useless repetitions. Indeed given a correct key with probability p the number of keys to try before it is found follows a geometric distribution with parameter p and thus has an expected value equal to $1/p$ with a variance of $\frac{1-p}{p^2}$. By contrast, for Algorithm 3, this number of keys to test is *at most* $\lfloor 1/p \rfloor$ (usually much less). Actually, our algorithm will output exactly n keys before the correct one if it is ranked in the n-th position, removing the variance issues of the probabilistic test. Also in practice, the probability-driven approach tends to lead to much more tests than optimal deterministic enumeration, as will be illustrated experimentally in the next section.

Next, and in terms of complexities, it is easy to see that the probability-driven algorithm can output new keys in constant time and has a very small memory requirement. The case of our deterministic enumeration algorithm is more difficult. The use of heaps for the frontier set and the recursive decomposition of the problem point towards a logarithmic time complexity. However, it appears from the experiments in the next section that the algorithm enumerates keys in amortized time close to constant. Summarizing, both methods lead to a linear time complexity in the total number of key candidates that are output, with the enumeration algorithm also requiring a sub-linear amount of memory.

As previously mentioned, an enumeration algorithm similar to ours was proposed in a paper by Pan, van Woudenberg, den Hartog and Witteman [21]. It also enumerates key candidates in optimal order, but the reduction rule 1 is not used, nor the recursive decomposition that allows us to efficiently apply rule 1 with more than two lists. Therefore, the frontier set of their algorithm is not reduced, and the memory requirements are much larger. In practice, since the main limitation for optimal key enumeration appears to be memory, this non-minimal version of enumeration does not allow an adversary to output a large number of key candidates. Moreover, handling a larger frontier set implies time complexity penalties, which makes our new algorithm faster than this previous one. Implementation results confirming these claims are given in the next section.

Finally, we note that another related problem is list decoding of convolutional codes through the Viterbi algorithm (see, e.g. [30]). However, and as previously mentioned, enumeration and decoding are not the same problems and the latter one only makes sense in the presence of redundancy, which does not exist when subkeys are independent of one another. An attempt at using such an algorithm would either lead to combinatorial explosion (i.e. 2^{128} possible states) for a deterministic version, or require very large amounts of memory when using approximate solutions such as beam search. Moreover, list-decoding algorithms are generally designed to output a small number of most likely candidates determined *a priori*, whereas we typically target the enumeration of 2^{32} or more master key candidates, continuing enumeration until the correct key is found.

Author Index