Thomas Eisenbarth
Erdinç Öztürk (Eds.)

LNCS 8898

Lightweight Cryptography for Security and Privacy

Third International Workshop, LightSec 2014
Istanbul, Turkey, September 1–2, 2014
Revised Selected Papers

Springer

Lecture Notes in Computer Science 8898

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

Editorial Board

David Hutchison
Lancaster University, Lancaster, UK
Takeo Kanade
Carnegie Mellon University, Pittsburgh, PA, USA
Josef Kittler
University of Surrey, Guildford, UK
Jon M. Kleinberg
Cornell University, Ithaca, NY, USA
Friedemann Mattern
ETH Zurich, Zürich, Switzerland
John C. Mitchell
Stanford University, Stanford, CA, USA
Moni Naor
Weizmann Institute of Science, Rehovot, Israel
C. Pandu Rangan
Indian Institute of Technology, Madras, India
Bernhard Steffen
TU Dortmund University, Dortmund, Germany
Demetri Terzopoulos
University of California, Los Angeles, CA, USA
Doug Tygar
University of California, Berkeley, CA, USA
Gerhard Weikum
Max Planck Institute for Informatics, Saarbrücken, Germany

More information about this series at http://www.springer.com/series/7410

Thomas Eisenbarth · Erdinç Öztürk (Eds.)

Lightweight Cryptography for Security and Privacy

Third International Workshop, LightSec 2014
Istanbul, Turkey, September 1–2, 2014
Revised Selected Papers

Springer

Editors
Thomas Eisenbarth
Worcester Polytechnic Institute
Worcester, MA
USA

Erdinç Öztürk
Istanbul Commerce University
Istanbul
Turkey

ISSN 0302-9743 ISSN 1611-3349 (electronic)
Lecture Notes in Computer Science
ISBN 978-3-319-16362-8 ISBN 978-3-319-16363-5 (eBook)
DOI 10.1007/978-3-319-16363-5

Library of Congress Control Number: 2015932981

LNCS Sublibrary: SL4 – Security and Cryptology

Springer Cham Heidelberg New York Dordrecht London
© Springer International Publishing Switzerland 2015
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
The publisher, the authors and the editors are safe to assume that the advice and information in this book are believed to be true and accurate at the date of publication. Neither the publisher nor the authors or the editors give a warranty, express or implied, with respect to the material contained herein or for any errors or omissions that may have been made.

Printed on acid-free paper

Springer International Publishing AG Switzerland is part of Springer Science+Business Media
(www.springer.com)

Preface

LightSec 2014 is the Third International Workshop on Lightweight Cryptography for Security and Privacy, which was held in Eminönü, Istanbul, Turkey, during September 1–2, 2014. The workshop was sponsored by TÜBİTAK BİLGEM UEKAE (The Scientific and Technological Research Council of Turkey, National Research Institute of Electronics and Cryptology) and held in cooperation with the International Association of Cryptologic Research (IACR).

The Program Committee (PC) consisted of 31 members representing 13 countries. There were 24 papers from 15 countries submitted to the workshop. Submitted papers were reviewed by the PC members themselves or by assigned subreviewers. Each submission was double-blind reviewed by at least three PC members and the submissions by PC members were assigned to 14 subreviewers. The vast majority of the papers were reviewed by four reviewers. Twelve of the papers were accepted for presentation at the workshop, whereas two of them were conditionally accepted. These two conditionally accepted papers were later withdrawn.

The program also included three excellent invited talks by experts in the field. The first talk was given by Tolga Acar from Microsoft Research titled "Selecting and Deploying Elliptic Curves in Security Protocols." The second talk was given by Guido Marco Bertoni from ST Microelectronics titled "Permutation-based encryption for lightweight applications." The final invited talk was delivered by Johann Heyszl from Fraunhofer AISEC titled "High-Resolution Magnetic Field Side-Channels and their Affect on Cryptographic Implementations."

We would like to thank all the people and organizations who contributed to making the workshop successful. First, we greatly appreciate the valuable work of the authors and we thank them for submitting their manuscripts to LightSec 2014. We are also grateful to the PC members and the External Reviewers whose admirable effort in reviewing the submissions definitely enhanced the scientific quality of the program. Thanks also to the invited speakers, Tolga Acar, Guido Marco Bertoni, and Johann Heyszl, for their willingness to participate in LightSec 2014. We would like to also thank Istanbul Chamber of Commerce and Istanbul Commerce University, who made this workshop possible by letting us use their facilities. We would like to thank Ali Boyacı and Serhan Yarkan from EE Engineering Department at Istanbul Commerce University for their admirable help in organizing the workshop. Last but not least, we would like to thank students of the EE Engineering Department at Istanbul Commerce University, for their help in running the workshop.

September 2014

Erdinç Öztürk
Thomas Eisenbarth

Organization

Organizing Committee

General Chair

Erdinç Öztürk	Istanbul Commerce University, Turkey

Program Committee

Program Chairs

Erdinç Öztürk	Istanbul Commerce University, Turkey
Thomas Eisenbarth	Worcester Polytechnic Institute, USA

Technical Program Committee

Tolga Acar	Microsoft Research, USA
Onur Aciicmez	Samsung, USA
Elena Andreeva	COSIC, KU Leuven, Belgium
Jean-Philippe Aumasson	Kudelski Security, Switzerland
Paulo Barreto	University of São Paulo, Brazil
Lejla Batina	Radboud University Nijmegen, The Netherlands
Guido Bertoni	ST Microelectronics, Italy
Andrey Bogdanov	Technical University of Denmark, Denmark
Elke De Mulder	Cryptography Research Inc., USA
Kris Gaj	George Mason University, USA
Berndt Gammel	Infineon Technologies, Germany
Shay Gueron	University of Haifa, Israel
Francisco Rodríguez Henríquez	CINVESTAV-IPN, Mexico
Pascal Junod	HEIG-VD, Switzerland
Jens-Peter Kaps	George Mason University, USA
Mehran Mozaffari Kermani	Rochester Institute of Technology, USA
Mehmet Sabir Kiraz	TÜBİTAK BİLGEM UEKAE, Turkey
Xuejia Lai	Shanghai Jiao Tong University, China
Albert Levi	Sabanci University, Turkey
Amir Moradi	Ruhr University Bochum Germany
Axel Poschmann	NXP Semiconductors, Germany
Francesco Regazzoni	ALaRI, Lugano, Switzerland
Arash Reyhani-Masoleh	University of Western Ontario, Canada
Erkay Savaş	Sabanci University, Turkey
Nitesh Saxena	University of Alabama at Birmingham, USA
Ali Aydin Selçuk	TOBB University of Economics and Technology, Turkey
Georg Sigl	Technische Universität München, Germany

Michael Tunstall	Cryptography Research Inc., USA
Meltem Sonmez Turan	National Institute of Standards and Technology, USA
Kerem Varici	Université Catholique de Louvain, Belgium
Amr Youssef	Concordia University, Canada

Steering Committee

Gildas Avoine	Université Catholique de Louvain, Belgium
Hüseyin Demirci	TÜBİTAK BİLGEM, Turkey
Orhun Kara	TÜBİTAK BİLGEM, Turkey
Erkay Savaş	Sabanci University Turkey
Ali Aydın Selçuk	TOBB University of Economics and Technology, Turkey
Berk Sunar	Worcester Polytechnic Institute, USA

Local Committee

Ali Boyacı	Istanbul Commerce University, Turkey
Serhan Yarkan	Istanbul Commerce University, Turkey

Additional Reviewers

S. Abhishek Anand
Shivam Bhasin
Begul Bilgin
Muhammed Ali Bingol
Joppe Bos
Joo Yeon Cho
Benedikt Driessen
Baris Ege
Jialin Huang
Süleyman Kardas
Thomas Korak
Wei Li
Suresh Limkar
John Michener
Manar Mohamed
Lan Nguyen
Kostas Papagiannopoulos
Roel Peeters
Gokay Saldamli
Fabrizio de Santis
Maliheh Shirvanian
Osmanbey Uzunkol
Rajesh Velegalati
Vincent Verneuil
MarkusWamser
Michael Weiner
Hong Xu
Panasayya Yalla
Greg Zaverucha

Sponsoring Institution

TÜBİTAK BİLGEM UEKAE (The Scientific and Technological Research Council of Turkey, National Research Institute of Electronics and Cryptology)

Powered by

INF Technology, Istanbul, Turkey

Invited Talks

Tolga Acar, Microsoft Research, USA

Selecting and Deploying Elliptic Curves in Security Protocols
The development and adoption of a cryptographic standard is a delicate endeavor with competing and conflicting actors, which becomes only harder with integration into security protocols some yet undefined. This talk looks at the use of Elliptic Curves (EC) in a sliver of pervasive security protocols. We cover NIST-defined ECs, impact of new information made available in the past couple of years, and current attempts to alleviate sometimes unsubstantiated yet valid concerns over these curves. This talk also presents an elliptic curve selection algorithm and its analysis from a performance and security perspective including rigid parameter generation, constant-time implementation, and exception-free scalar multiplication.

Guido Marco Bertoni, ST Microelectronics, Italy

Permutation-based encryption for lightweight applications
In the recent years we have seen a rapid development of cryptographic primitives based on permutations. The talk gives an overview on how you can easily build hash functions, stream ciphers, PRNGs, authenticated encryption and other constructions starting from a fixed-width permutation. This flexibility can be particular useful in resource-constrained applications, basically a single primitive can satisfy all the security needs typically requested to symmetric key primitives. Finally there will be the introduction of Ketje, a lightweight authenticated encryption developed in collaboration with Joan Daemen, Michael Peeters, Gilles Van Assche and Ronny Van Keer.

Johann Heyszl, Fraunhofer AISEC, Germany

High-Resolution Magnetic Field Side-Channels & their Affect on Cryptographic Implementations
The last years have again seen many new developments in the field of side-channel analysis. Partly, new insights are driven by side-channel measurement equipment which becomes increasingly sophisticated due to the fact that respective devices are readily available to academics, as well as to industry and potential attackers. This talk discusses the impact of available high-resolution measurement equipment to measure magnetic fields on implementations of cryptographic algorithms. The progress in this segment of side-channel analysis affects different kinds of cryptographic implementations including light-weight implementations of elliptic curve cryptography, symmetric cryptographic algorithms, physical unclonable functions as well as new attempts to achieve leakage resilience for block ciphers by special constructions.

Contents

Efficient Implementations and Designs

The SIMON and SPECK Block Ciphers on AVR 8-Bit Microcontrollers. 3
Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers

The Multiplicative Complexity of Boolean Functions on Four and Five Variables 21
Meltem Turan Sönmez and René Peralta

A Flexible and Compact Hardware Architecture for the SIMON Block Cipher 34
Ege Gulcan, Aydin Aysu, and Patrick Schaumont

AES Smaller Than S-Box: Minimalism in Software Design on Low End Microcontrollers 51
Mitsuru Matsui and Yumiko Murakami

Attacks

Differential Factors: Improved Attacks on SERPENT 69
Cihangir Tezcan and Ferruh Özbudak

Ciphertext-Only Fault Attacks on PRESENT 85
Fabrizio De Santis, Oscar M. Guillen, Ermin Sakic, and Georg Sigl

Relating Undisturbed Bits to Other Properties of Substitution Boxes 109
Rusydi H. Makarim and Cihangir Tezcan

Differential Sieving for 2-Step Matching Meet-in-the-Middle Attack with Application to LBlock 126
Riham AlTawy and Amr M. Youssef

Match Box Meet-in-the-Middle Attacks on the SIMON Family of Block Ciphers. 140
Ling Song, Lei Hu, Bingke Ma, and Danping Shi

Protocols

A Provably Secure Offline RFID Yoking-Proof Protocol with Anonymity . . . 155
Daisuke Moriyama

Author Index 169

Efficient Implementations and Designs

The Simon and Speck Block Ciphers on AVR 8-Bit Microcontrollers

Ray Beaulieu, Douglas Shors, Jason Smith(✉), Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers

National Security Agency, 9800 Savage Road, Fort Meade, MD 20755, USA
{rabeaul,djshors,jksmit3,sgtreat,beweeks,lrwinge}@tycho.ncsc.mil

Abstract. The last several years have witnessed a surge of activity in lightweight cryptographic design. Many lightweight block ciphers have been proposed, targeted mostly at hardware applications. Typically software performance has not been a priority, and consequently software performance for many of these algorithms is unexceptional. Simon and Speck are lightweight block cipher families developed by the U.S. National Security Agency for high performance in constrained hardware *and* software environments. In this paper, we discuss software performance and demonstrate how to achieve high performance implementations of Simon and Speck on the AVR family of 8-bit microcontrollers. Both ciphers compare favorably to other lightweight block ciphers on this platform. Indeed, Speck seems to have better overall performance than any existing block cipher — lightweight or not.

Keywords: Simon · Speck · Block cipher · Lightweight · Cryptography · Microcontroller · Wireless sensor · AVR

1 Introduction

The field of lightweight cryptography is evolving rapidly in order to meet future needs, where interconnected, highly constrained hardware and software devices are expected to proliferate.

Over the last several years the international cryptographic community has made significant strides in the development of lightweight block ciphers. There are now more than a dozen to choose from, including Present [6], Clefia [20], Twine [21], Hight [14], Klein [12], LBlock [25], Piccolo [19], Prince [7], LED [13], Katan [9], TEA [23], and ITUbee [16]. Typically, a design will offer improved performance on some platform (e.g., ASIC, FPGA, microcontroller, microprocessor) relative to its predecessors. Unfortunately, most have good performance in hardware or software, but not both. This is likely to cause problems when communication is required across a network consisting of many disparate (hardware- and software-based) devices, and the highly-connected nature of

The rights of this work are transferred to the extent transferable according to title 17 § 105 U.S.C.

© Springer International Publishing Switzerland 2015 (outside the US)
T. Eisenbarth and E. Öztürk (Eds.): LightSec 2014, LNCS 8898, pp. 3–20, 2015.
DOI: 10.1007/978-3-319-16363-5_1

developing technologies will likely lead to many applications of this type. Ideally, lightweight cryptography should be *light* on a variety of hardware and software platforms. Cryptography of this sort is relatively difficult to create and such general-purpose lightweight designs are rare.

Recently, the U.S. National Security Agency (NSA) introduced two general-purpose lightweight block cipher families, Simon and Speck [2], each offering high performance in *both* hardware and software. In hardware, Simon and Speck have among the smallest reported implementations of existing block ciphers with a flexible key.[1] Unlike most hardware-oriented lightweight block ciphers, Simon and Speck also have excellent software performance.

In this paper, we focus on the software performance of Simon and Speck. Specifically, we show how to create high performance implementations of Simon and Speck on 8-bit Atmel AVR microcontrollers. This paper should be useful for developers trying to implement Simon or Speck for their own applications and interesting for cryptographic designers wishing to see how certain design decisions affect performance.

2 The Simon and Speck Block Ciphers

Simon and Speck are block cipher families, each comprising ten block ciphers with differing block and key sizes to closely fit application requirements. Table 1 shows the available block and key sizes for Simon and Speck.

Table 1. Simon and Speck parameters.

Block size	Key sizes
32	64
48	72, 96
64	96, 128
96	96, 144
128	128, 192, 256

The Simon (resp. Speck) block cipher with a $2n$-bit block and wn-bit key is denoted by Simon $2n/wn$ (resp. Speck $2n/wn$). Together, the round functions of the two algorithms make use of the following operations on n-bit words:

- bitwise `XOR`, $\oplus$,
- bitwise `AND`, $\&$,
- left circular shift, S^j, by j bits,
- right circular shift, S^{-j}, by j bits, and
- modular addition, $+$.

[1] Simon 64/96 and Speck 64/96, for example, have implementations requiring just 809 and 860 gate equivalents, respectively. Some block ciphers, like KTANTAN [9], have a fixed key and so do not require flip-flops to store it. Such algorithms can have smaller hardware implementations than Simon or Speck, but not allowing keys to change contracts the application space, and can lead to security issues [22].

For $k \in \mathrm{GF}(2)^n$, the key-dependent Simon $2n$ round function is the two-stage Feistel map $R_k\colon \mathrm{GF}(2)^n \times \mathrm{GF}(2)^n \to \mathrm{GF}(2)^n \times \mathrm{GF}(2)^n$ defined by

$$R_k(x, y) = (y \oplus f(x) \oplus k,\ x),$$

where $f(x) = (Sx \,\&\, S^8x) \oplus S^2x$ and k is the round key. The key-dependent Speck $2n$ round function is the map $R_k\colon \mathrm{GF}(2)^n \times \mathrm{GF}(2)^n \to \mathrm{GF}(2)^n \times \mathrm{GF}(2)^n$ defined by

$$R_k(x, y) = ((S^{-\alpha}x + y) \oplus k,\ S^{\beta}y \oplus (S^{-\alpha}x + y) \oplus k),$$

with rotation amounts $\alpha = 7$ and $\beta = 2$ if $n = 16$ (block size = 32) and $\alpha = 8$ and $\beta = 3$, otherwise. A description of the key schedules, not necessary for this paper, can be found in the Simon and Speck specification paper [2].

3 AVR Implementations of Simon and Speck

We have implemented Simon and Speck on the Atmel ATmega128, a low-power 8-bit microcontroller with 128K bytes of programmable flash memory, 4K bytes of SRAM, and thirty-two 8-bit general purpose registers. To achieve optimal performance, we have coded Simon and Speck in assembly. The simplicity of the algorithms makes this easy to do, with the help of the Atmel AVR 8-bit instruction set manual [1].

Our assembly code was written and compiled using Atmel Studio 6.0. Cycle counts, code sizes and RAM usage were also determined using this tool. Our complete set of performance results is provided in Appendix A.

For Simon and Speck, three implementations were developed, none of which included the decryption algorithm.[2]

(1) **RAM-minimizing implementations.** These implementations avoid the use of RAM to store round keys by including the pre-expanded round keys in the flash program memory. No key schedule is included for updating this expanded key, making these implementations suitable for applications where the key is static.

(2) **High-throughput/low-energy implementations.** These implementations include the key schedule and unroll enough copies of the round function in the encryption routine to achieve a throughput within about 3% of a fully-unrolled implementation. The key, stored in flash, is used to generate the round keys which are subsequently stored in RAM.

[2] This is because one is likely to use encrypt-only modes in lightweight cryptography. But the techniques discussed here should serve as a starting point for other kinds of implementations, useful for a broad range of applications. Regarding decryption functionality, we note that the Simon and Speck encryption and decryption algorithms consume similar resources and are easy to implement. Simon, in particular, has a decryption algorithm that is closely related to the encryption algorithm, and so little additional code is necessary to enable decryption.

(3) **Flash-minimizing implementations.** The key schedule is included here. Space limitations mean we can only provide an incomplete description of these implementations. However, it should be noted that the previous two types of implementations already have very modest code sizes.

In almost all cases, the registers hold the intermediate ciphertext, the currently used round key, and any data needed to carry out the computation of a round. For the algorithms with the 128-bit block and 192-bit or 256-bit key, there may not be enough registers to hold all of this information, and so it may be necessary to store one or two 64-bit words of round key in RAM. Additional modifications were necessary for the 128-bit SIMON block ciphers.

We describe our various assembly implementations of SIMON 64/128 and SPECK 64/128 on an AVR 8-bit microcontroller using pseudo assembly code. When it is not obvious, we show how to translate the pseudocode into actual assembly instructions. Although our implementations were aimed at the ATmega128 microcontroller, the same techniques are applicable to other AVR microcontrollers, e.g., the ATtiny line (except when noted).

4 SIMON AVR Implementations

We describe various implementations of SIMON 64/128 on the ATmega128 microcontroller. Implementations of the other algorithms in the family are similar.

The ATmega128 has thirty-two 8-bit registers. For the purpose of exposition, we regard a sequence of four such registers as a 32-bit register, denoted by a capital letter such as X, Y, K, etc. For instance, $X = (X_3, X_2, X_1, X_0)$ identifies the 32-bit register X as a sequence of four 8-bit registers. We denote the contents of these 32-bit registers by lower case letters x, y, k, etc.

At the start of the encryption process, we assume a 64-bit plaintext pair (x, y) resides in RAM, and is immediately loaded into an (X, Y) register pair for processing. After 44 encryption rounds, the resulting ciphertext is stored in RAM. The encryption cost is the number of cycles needed to transform the plaintext into the ciphertext, including any loading of plaintext/ciphertext into or out of RAM.[3] Our performance numbers do not count the cost of RAM used to hold the plaintext, ciphertext, or key but do include RAM used for temporary storage (e.g., stack memory) and RAM to hold the round keys.

Except for the small code size implementations of SIMON, developers should avoid the use of any loops or branching within a round, since this can significantly degrade overall performance and does not greatly reduce code size.

4.1 A Minimal RAM Implementation of SIMON

Here we assume the round keys have been pre-expanded (by an external device) and stored in flash. When a round key is required, it is loaded from flash directly into a register. Loading a byte from flash consumes three cycles.

[3] The SIMON and SPECK specification paper [2] did not count these cycles required for loading, although it seems proper to do so. The current performance numbers include these costs.

We begin by describing the `Rotate` operation, which performs a left circular shift by one. Let $X = (X_3, X_2, X_1, X_0)$ and let $Z_0 = 0$. The 8-bit register Z_0 should be cleared and reserved at the start of encryption. The 1-bit rotation of the 32-bit register X is carried out using AVR's logical shift left (`LSL`), rotate left through carry (`ROL`), and add with carry (`ADC`) instructions, as follows.

(1) $X_0 \leftarrow \texttt{LSL}(X_0)$ (logical shift left)
(2) $X_1 \leftarrow \texttt{ROL}(X_1)$ (rotate left through carry)
(3) $X_2 \leftarrow \texttt{ROL}(X_2)$ (rotate left through carry)
(4) $X_3 \leftarrow \texttt{ROL}(X_3)$ (rotate left through carry)
(5) $X_0 \leftarrow \texttt{ADC}(X_0, Z_0)$ (add with carry)

In general, this *standard* approach to performing a rotation requires $n + 1$ cycles on an n byte word. Rotations by more than one bit can be achieved by repeated one-bit rotations, though this is not always the most efficient approach.

Note that the rotation by 8 is free, because it's just a byte permutation.[4] The rotations by 1 and 2 are also inexpensive. Our pseudo instruction `Move` is shorthand for two AVR `MOVW` instructions, each of which copies two 8-bit words from one register to another in a single cycle. In order to use the `MOVW` instruction, the 8-bit registers must be properly aligned [1]. Our other mnemonics are also shorthand for readily apparent AVR instructions. Table 2 describes how to implement a round of SIMON.

Table 2. Low-RAM software implementation of the SIMON round function.

Mnemonic	Operation	Register Contents	Cycles
`Load`	$K \leftarrow k$	$K = k$	12
`XOR`	$K \leftarrow K \oplus Y$	$K = y \oplus k$	4
`Move`	$Y \leftarrow X$	$Y = x$	2
`Rotate`	$X \leftarrow S^1(X)$	$X = S^1(x)$	5
`Move`	$T \leftarrow X$	$T = S^1(x)$	2
`And`	$T \leftarrow T \,\&\, S^8(Y)$	$T = S^1(x) \,\&\, S^8(x)$	4
`Rotate`	$X \leftarrow S^1(X)$	$X = S^2(x)$	5
`XOR`	$X \leftarrow X \oplus T$	$X = S^2(x) \oplus S^1(x) \,\&\, S^8(x)$	4
`XOR`	$X \leftarrow X \oplus K$	$X = y \oplus S^2(x) \oplus S^1(x) \,\&\, S^8(x) \oplus k$	4

The basic code for a round executes in 42 cycles. For the minimal RAM implementation, this code is put in a loop which has a 3 cycle per round overhead. Since SIMON 64/128 requires 44 rounds, an encryption will take about $44 \cdot (42 + 3) = 1980$ cycles. There are ten cycles needed for setup, a subroutine call, and a return plus an additional 32 cycles to load the plaintext into registers

[4] This rotation is also easily implemented (but not for free) on some common 16-bit microcontrollers, like the MSP430, and using x86 SSE instructions (where no rotate is available but a byte permutation operation is).

(2 cycles/byte) and load the resulting ciphertext back into RAM (2 cycles/byte). Altogether, SIMON 64/128 has an encryption cost of 253 cycles/byte.

In terms of code size, each instruction, with the exception of the `Load` instruction, takes twice as many bytes to store in flash as the number of cycles it takes to execute. The four byte `Load` instruction, in Table 2, consumes 8 bytes of flash. The expanded key is stored in $44 \cdot 4 = 176$ bytes of flash. So the total amount of flash used is around $68 + 44 \cdot 4 = 244$ bytes. More bytes are required for the counter, loading plaintext, storing ciphertext and other miscellaneous overhead so the actual value is 290 bytes. No RAM was used for this implementation.

4.2 A High-Throughput/Low-Energy Implementation of SIMON

High-throughput implementations are useful when encrypting multiple blocks of data. The round keys for such an implementation can initially be stored in flash, as with our low-RAM implementations, or they can be generated using the key schedule. In either case, the encryption process begins by placing all of the round keys into RAM, where they are held until all of the blocks in a stream of data have been encrypted. For our implementations, we used the key schedule to generate the round keys, but for our timings we have not included the time to generate and store these since this *setup* time is assumed to be small compared to the time required to encrypt the data stream.

For SIMON 64/128, we unrolled four rounds of the code and iterated this 11 times to carry out the 44 round encryption. Because the key is now loaded from RAM, the `Load` requires only 8 cycles instead of the 12 which were needed when loading from flash. Due to the larger code size, a four cycle overhead for each update of the loop counter (as opposed to three cycles) is required, together with an additional ten cycles as described earlier and 32 more cycles for loading plaintext in registers and storing the ciphertext in RAM. The amount of unrolling was calibrated to achieve throughput to within 3 % of the fastest (fully unrolled) implementation, avoiding large increases in code size with minor improvements in throughput. The cost of this implementation is 221 cycles/byte.

The code size for the encryption algorithm is about $68 \cdot 4 = 272$ bytes. Together with the key schedule and other overhead, this implementation of SIMON 64/128 uses 436 bytes of flash. Because all of the round keys are placed in RAM, $44 \cdot 4 = 176$ bytes of RAM are also required.

4.3 A Minimal Flash Implementation of SIMON

To reduce the already small size of SIMON, we implemented several frequently used operations as subroutines. These included the `XOR`, the 1-bit `Rotate` and the `Move` instructions. Doing this, we saved a dozen bytes for SIMON 64, and more than 50 bytes for SIMON 128. For SIMON 32, these techniques ended up not saving any space and degraded throughput, so we did not use them.

SIMON 64/128, in particular, can be implemented using 240 bytes of flash and with $4 \cdot 44 = 176$ bytes of RAM to store the round keys. We note that for an additional 28 bytes of flash, decrytpion capability can be added: To decrypt,

one uses the round keys in reverse order, loads the swapped ciphertext words into the encryption round function, and reads out the plaintext words with a similar swap. The swapped loading and output (together with the regular loading and output) can be done with 8 additional bytes of code each, and the round keys can be generated and stored in normal or reverse order with 12 additional bytes of code, for a total of $8 + 8 + 12 = 28$ bytes.

5 SPECK AVR Implementations

In this section, we describe the same types of implementations that we previously developed for the SIMON 64/128 block cipher. We stress that most of the implementation techniques described here apply immediately to the other members of the SPECK family.

Using the same notation as in SIMON, we assume a 64-bit plaintext pair (x, y) resides in RAM. This is immediately loaded into the 64-bit register pair (X, Y). After 27 encryption rounds, the resulting ciphertext is loaded into RAM. The encryption cost is the number of cycles required to load the plaintext into registers, transform the plaintext into the ciphertext, and store the ciphertext in RAM.

For all of our implementations, we avoid the use of any loops or branching within a round. This has the effect of making our code slightly larger but significantly improves the overall performance.

For SIMON, our main tool for trading off code size for throughput was to partially unroll loops. It turns out that for SPECK, there is more opportunity to tune the implementation (for example using multiply instructions to accomplish the rotation by 3 or by allowing a round to end with plaintext and key words in the *wrong* places) to make it smaller or faster. Consequently, we will spend a little more time describing SPECK implementations.

5.1 A Low-RAM SPECK Implementation

Here we assume the round keys have been pre-expanded and stored in flash. When a round key k is required, it is loaded from flash directly into a register with a cost of 12 cycles. Table 3 describes how to implement the SPECK round.

Table 3. Low-RAM software implementation of the SPECK round function.

Mnemonic	Operation	Register Contents	Cycles
`Load`	$K \leftarrow k$	$K = k$	12
`Add`	$X \leftarrow S^8(S^{-8}(X) + Y)$	$X = S^8(S^{-8}(x) + y)$	4
`XOR`	$K \leftarrow K \oplus S^{-8}(X)$	$K = (S^{-8}(x) + y) \oplus k$	4
`Rotate`	$Y \leftarrow S^3(Y)$	$Y = S^3(y)$	15
`XOR`	$Y \leftarrow Y \oplus K$	$Y = S^3(y) \oplus (S^{-8}(x) + y) \oplus k$	4
`Move`	$X \leftarrow K$	$X = (S^{-8}(x) + y) \oplus k$	2

Note that the rotation by 8 is free, as we noted in the Simon discussion. The `Rotate` and `Move` instructions were described earlier for Simon. To be clear, we describe the `Add` operation in greater detail. If $X = (X_3, X_2, X_1, X_0)$ and $Y = (Y_3, Y_2, Y_1, Y_0)$, then addition, `Add`, is carried out using the following AVR instructions in the given order.

$$\begin{array}{lll} (1) & X_1 \leftarrow X_1 + Y_0 & \text{(add without carry, ADD)} \\ (2) & X_2 \leftarrow X_2 + Y_1 & \text{(add with carry, ADC)} \\ (3) & X_3 \leftarrow X_3 + Y_2 & \text{(add with carry, ADC)} \\ (4) & X_0 \leftarrow X_0 + Y_3 & \text{(add with carry, ADC)} \end{array}$$

Our basic code for a round executes in 41 cycles, and this code is iterated 27 times in a loop to produce the ciphertext. The loop has an overhead of 3 cycles per round, and since Speck 64/128 requires 27 rounds, an encryption takes about $27 \cdot (41 + 3) = 1188$ cycles. An additional 10 more cycles for overhead (3 cycles for setup, 3 for a subroutine call, and 4 for a return) plus 32 cycles for loading plaintext from RAM into registers and storing ciphertext back into RAM, brings the total number of cycles for an encryption to 1230, which translates to $1230/8 \approx 154$ cycles/byte.

As we noted in the Simon discussion, each instruction, with the exception of the `Load` instruction, takes twice as many bytes to store in flash as the number of cycles it takes to execute. The `Load` instruction requires 8 bytes of flash.

The Speck round keys consume $27 \cdot 4 = 108$ bytes of flash. The total amount of flash required for the round and expanded key is $66 + 108 = 174$ bytes. Additional bytes, required for the counter, loading and storing plaintext and ciphertext and other overhead, brings the total to 218 bytes. No RAM was required for this implementation.

5.2 A Faster Low-RAM Speck Implementation

If a higher-throughput/lower-energy implementation is required, we can easily modify the implementation that we just described to obtain one which encrypts at a rate of 142 cycles/byte using around 342 bytes of flash and no RAM.

This is done by iterating two rounds multiple times. Two rounds can be implemented without the use of the `Move` that appeared in Table 3, thereby saving two cycles per round. The two rounds are iterated in a loop 13 times (for a total of 26 rounds) and a final single round of code as described in Table 3 is used for the 27th round. For members of the family requiring an even number of rounds (like Speck 64/96) this extra code for the final round is not required. For Speck 64/128, the number of cycles for a complete encryption is around $2 \cdot 39 \cdot 13 + 41 + 13 \cdot 3 + 10 + 32 = 1136$, or about 142 cycles/byte. About 342 bytes of flash are required.

The pseudocode for the two rounds is shown in Table 4. There, x_1 and y_1 are the values of the input to the second round, stored in the K and Y registers and l is a new round key which is loaded into the X register. The output to the second round is again stored in the *proper* registers, i.e., the X and Y registers.

Table 4. A faster, low-RAM software implementation of two rounds of SPECK.

Mnemonic	Operation	Register Contents	Cycles
`Load`	$K \leftarrow k$	$K = k$	12
`Add`	$X \leftarrow S^8(S^{-8}(X) + Y)$	$X = S^8(S^{-8}(x) + y)$	4
`XOR`	$K \leftarrow K \oplus S^{-8}(X)$	$K = (S^{-8}(x) + y) \oplus k = x_1$	4
`Rotate`	$Y \leftarrow S^3(Y)$	$Y = S^3(y)$	15
`XOR`	$Y \leftarrow Y \oplus K$	$Y = S^3(y) \oplus (S^{-8}(x) + y) \oplus k = y_1$	4
`Load`	$X \leftarrow l$	$X = l$	12
`Add`	$K \leftarrow S^8(S^{-8}(K) + Y)$	$K = S^8(S^{-8}(x_1) + y_1)$	4
`XOR`	$X \leftarrow X \oplus S^{-8}(K)$	$X = (S^{-8}(x_1) + y_1) \oplus l$	4
`Rotate`	$Y \leftarrow S^3(Y)$	$Y = S^3(y_1)$	15
`XOR`	$Y \leftarrow Y \oplus X$	$Y = S^3(y_1) \oplus (S^{-8}(x_1) + y_1) \oplus l$	4

5.3 A High-Throughput/Low-Energy SPECK Implementation

As in the corresponding SIMON implementations, the SPECK encryption process begins by placing all of the round keys into RAM and holding them there until the data stream has been encrypted. Although we used the key schedule to generate the round keys, the round keys could also be pre-computed and stored in flash before loading them into RAM. For our timings, we have not included the time to generate the round keys and store them in RAM.

The easiest way to obtain a fast encryption algorithm is just to modify the fast, low-RAM implementation described previously using the code found in Table 4, but now loading the round keys from RAM instead of from flash. The cost of loading four bytes from RAM is 8 cycles, as opposed to 12 if we load from flash. So the total cost to encrypt a block of data is $2 \cdot 35 \cdot 13 + 37 + 13 \cdot 4 + 10 + 32 = 1041$ cycles, or about 130 cycles/byte. Not including the key schedule, the code will occupy about 232 bytes. Taking into account the key schedule, the code size is about 352 bytes and the implementation uses 108 bytes of RAM.

We could speed this up even more, at the expense of using more flash, by unrolling four rounds of code instead of just two. If we unroll all of the rounds we get a heavy implementation, in terms of flash, that encrypts a block in just 123 cycles/byte. If this sort of implementation is appealing but the flash usage is not, there is another way to get the same throughput using significantly less flash. However, the technique only works on those AVR microcontrollers, such as the ATmega128, which include the AVR unsigned multiplication instruction `MUL`. The ATtiny line does not have the `MUL` instruction.

We now describe how to do the 3-bit rotation operation `Rotate` in 14 cycles using 20 bytes of flash with the AVR `MUL` instruction. This should be compared to the in-place rotation used earlier, which required 15 cycles and 30 bytes.

The `MUL` instruction operates on two 8-bit registers containing unsigned numbers and produces the 16-bit unsigned product in 2 cycles. The result is always placed in the AVR register pair (R_1, R_0), the low bits in R_0 and the high bits

in R_1. Letting $Y = (Y_3, Y_2, Y_1, Y_0)$ and $X = (X_3, X_2, X_1, X_0)$, the `Rotate` operation, $X \leftarrow S^3(Y)$, is implemented as follows:

$$\begin{array}{rrcl}
(1) & (R_1, R_0) & \leftarrow & \texttt{MUL}(Y_0, 8) \\
(2) & (X_1, X_0) & \leftarrow & (R_1, R_0) \\
(3) & (R_1, R_0) & \leftarrow & \texttt{MUL}(Y_2, 8) \\
(4) & (X_3, X_2) & \leftarrow & (R_1, R_0) \\
(5) & (R_1, R_0) & \leftarrow & \texttt{MUL}(Y_1, 8)
\end{array}$$

$$\begin{array}{rrcl}
(6) & X_1 & \leftarrow & X_1 \oplus R_0 \\
(7) & X_2 & \leftarrow & X_2 \oplus R_1 \\
(8) & (R_1, R_0) & \leftarrow & \texttt{MUL}(Y_3, 8) \\
(9) & X_3 & \leftarrow & X_3 \oplus R_0 \\
(10) & X_0 & \leftarrow & X_0 \oplus R_1
\end{array}$$

Here, 8 represents an 8-bit register with the value 8 stored in it. This register should be initialized and reserved at the start of the encryption process. It should be noted that this rotate algorithm is not in-place. Additionally, for a w-byte word, the performance can be worse than for the standard approach: For $w = 2j$, the running time is $7j$ cycles for this technique versus $3\cdot(2j+1) = 6j+3$ cycles for the standard approach. Thus, for the highest speed implementations, the technique is advantageous for Speck 48 and Speck 64. For Speck 96 it doesn't increase throughput, but it can still significantly decrease the flash usage, and so we adopt it. For Speck 128, the method will actually decrease throughput. The code for a round of Speck is provided in Table 5.

Table 5. A high-throughput/low-power round implementation of Speck

Mnemonic	Operation	Register Contents	Cycles
`Load`	$K \leftarrow k$	$K = k$	8
`Add`	$X \leftarrow S^8(S^{-8}(X) + Y)$	$X = S^8(S^{-8}(x) + y)$	4
`XOR`	$K \leftarrow S^{-8}(X) \oplus K$	$K = (S^{-8}(x) + y) \oplus k$	4
`Rotate`	$X \leftarrow S^3(Y)$	$X = S^3(y)$	14
`XOR`	$X \leftarrow X \oplus K$	$X = (S^{-8}(x) + y) \oplus k \oplus S^3(y)$	4

The output of the first round is stored in the (K, X) register pair. In order to get the output in the proper, X and Y registers, we need to move X into Y and K into X. However, if we don't want to incur additional cycles by doing this, then we can proceed in a manner similar to the fast, low-RAM implementation by iterating three consecutive rounds, all identical up to a relabeling of registers. The final output after the third round will be back in the (X, Y) register pair. The code for doing this is shown in Table 6.

If we iterate three rounds in a loop nine times as just described, Speck 64/128 performs a full encryption in around $34 \cdot 27 + 4 \cdot 9 + 10 + 32 = 996$ cycles, or about 125 cycles/byte and, including the key schedule, uses 316 bytes of flash.

Table 6. A high-throughput/low-energy three round implementation of Speck

Mnemonic	Operation	Register Contents	Cycles
`Load`	$K \leftarrow k$	$K = k$	8
`Add`	$X \leftarrow S^8(S^{-8}(X) + Y)$	$X = S^8(S^{-8}(x) + y)$	4
`XOR`	$K \leftarrow S^{-8}(X) \oplus K$	$K = (S^{-8}(x) + y) \oplus k = x_1$	4
`Rotate`	$X \leftarrow S^3(Y)$	$X = S^3(y)$	14
`XOR`	$X \leftarrow X \oplus K$	$X = S^3(y) \oplus (S^{-8}(x) + y) \oplus k = y_1$	4
`Load`	$Y \leftarrow l$	$Y = l$	8
`Add`	$K \leftarrow S^8(S^{-8}(K) + X)$	$K = S^8(S^{-8}(x_1) + y_1)$	4
`XOR`	$Y \leftarrow S^{-8}(K) \oplus Y$	$Y = (S^{-8}(x_1) + y_1) \oplus l = x_2$	4
`Rotate`	$K \leftarrow S^3(X)$	$K = S^3(y_1)$	14
`XOR`	$K \leftarrow K \oplus Y$	$K = S^3(y_1) \oplus (S^{-8}(x_1) + y_1) \oplus l = y_2$	4
`Load`	$X \leftarrow m$	$X = m$	8
`Add`	$Y \leftarrow S^8(S^{-8}(Y) + K)$	$Y = S^8(S^{-8}(x_2) + y_2)$	4
`XOR`	$X \leftarrow S^{-8}(Y) \oplus X$	$X = (S^{-8}(x_2) + y_2) \oplus m = x_3$	4
`Rotate`	$Y \leftarrow S^3(K)$	$Y = S^3(y_2)$	14
`XOR`	$Y \leftarrow Y \oplus X$	$Y = S^3(y_2) \oplus (S^{-8}(x_2) + y_2) \oplus m = y_3$	4

The high-throughput data for Speck 64/128, found in Table 7 of Appendix A, was obtained by unrolling 6 rounds instead of 3 in order to get within 3% of the fastest, fully unrolled, implementation. That gave us a 122 cycles/byte implementation but required about twice the flash.

5.4 A Small Flash Speck Implementation

Speck can be implemented to have small code size. We have not attempted to minimize the code size without regard to the throughput, so smaller implementations are possible. The primary savings in space was achieved by implementing the Speck round function as a subroutine. In this way, both the key schedule and encryption function could use it without having to duplicate code.

6 Cipher Comparisons

Although it is not the primary purpose of this paper, we shall compare the performance of Simon and Speck with several other block ciphers, and with a few stream ciphers. Comparisons along these lines can already be found in the Simon and Speck specification paper [2]. We have not endeavored to implement all of the discussed ciphers from scratch since this was outside the scope of our paper. Instead, when possible, we (or others) modified the best implementations available from existing sources so that they fit within our framework. Our comparison data and a further discussion of our methodology can be found in Appendix B.

For brevity, we only included algorithms with an encryption cost of 1000 cycles/byte or less. For this reason, or because the data was unavailable, well-known lightweight block ciphers such as Present, Clefia, Katan, etc., are not listed. We use a performance efficiency measure, *rank*, that is similar to a commonly used metric (see [16,18,21]), proportional to throughput divided by memory usage.

Among all block ciphers, Speck ranks in the top spot for every block and key size which it supports. Except for the 128-bit block size, Simon ranks second for all block and key sizes. Among the 64-bit block ciphers, Hight has respectable performance on this platform, ranking higher than AES. Although Twine ranks lower than AES, its performance is reasonable and, additionally, it has very good hardware performance, making it one of the better lightweight designs. Some software-based designs, like ITUbee, IDEA and TEA have poorer performance than AES and are not compensated by particularly lightweight hardware implementations. Klein, a hardware-based design, is slow and has the least competitive overall performance among the 64-bit block ciphers in our comparison.

Not surprisingly, AES-128 has very good performance on this platform, although for the same block and key size, Speck has about twice the performance. For the same key size but with a 64-bit block size, Simon and Speck achieve two and four times better overall performance, respectively, than AES. A few of the block ciphers in our comparison could not outperform AES, even though they had smaller block and key sizes.

If an application requires high speed, and memory usage is not a priority, AES has the fastest implementation (using 1912 bytes of flash, 432 bytes RAM) among all block ciphers with a 128-bit block and key that we are aware of, with a cost of just 125 cycles/byte [8]. The closest AES competitor is Speck 128/128, with a cost of 138 cycles/byte for a fully unrolled implementation. Since speed is correlated with energy consumption, AES-128 may be a better choice in energy critical applications than Speck 128/128 on this platform.[5] However, if a 128-bit block is not required, as we might expect for many applications on an 8-bit microcontroller, then a more energy efficient solution (using 628 bytes of flash, 108 bytes RAM) is Speck 64/128 with the same key size as AES-128 and an encryption cost of just 122 cycles/byte, or Speck 64/96 with a cost of 118 cycles/byte.

We have also compared three of the four Profile I (software) eSTREAM competition finalists, Salsa 20/12 [4], Sosemanuk [3], and HC-128 [24] since there is a general perception that well-designed stream ciphers must have better performance than block ciphers.[6] Interestingly, all of the stream ciphers are less efficient than the majority of the block ciphers listed. For high-speed/low-energy applications, if memory is not a concern, only Sosemanuk can beat the fastest implementations of AES and Speck.

[5] We do not know, for a fact, that the high-speed AES implementations, which require frequent calls to RAM, are more energy efficient than the high-speed Speck implementations which use mostly register-to-register operations.

[6] No data for the other finalist, Rabbit [5], was available.

A SIMON and SPECK AVR Performance

In this section we present the results of our three AVR implementations of SIMON (Table 8) and SPECK (Table 7). The headings of each column indicate the block size/key size in bits. The first three rows of data correspond to the low flash implementation, the next three rows to the low RAM implementation and the last three rows are for the low cost (i.e., high-speed/low-energy) implementation. RAM and flash are measured in bytes and cost is measured in cycles/byte. For none of the algorithms is any sort of functionality for the decryption operator included, although for SIMON decryption is essentially the same as encryption, and SPECK decryption uses about the same amount of resources as SPECK encryption.

Table 7. SPECK AVR implementation data.

	32/64	48/72	48/96	64/96	64/128	96/96	96/144	128/128	128/192	128/256
flash	**118**	**150**	**156**	**184**	**192**	**218**	**254**	**278**	**330**	**348**
RAM	44	69	72	108	112	174	180	264	272	280
cost	171	149	155	158	164	154	159	169	174	179
flash	108	154	158	214	218	324	330	460	468	476
RAM	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
cost	144	134	140	148	154	152	157	171	176	181
flash	440	556	586	588	628	502	624	452	632	522
RAM	44	66	69	104	108	168	174	256	272	288
cost	**114**	**104**	**108**	**118**	**122**	**127**	**131**	**143**	**147**	**151**

Table 8. SIMON AVR implementation data.

	32/64	48/72	48/96	64/96	64/128	96/96	96/144	128/128	128/192	128/256
flash	**152**	**190**	**202**	**230**	**240**	**304**	**304**	**392**	**392**	**404**
RAM	64	108	108	168	176	312	324	544	552	576
cost	193	206	206	222	232	262	272	346	351	366
flash	130	190	190	282	290	474	486	760	768	792
RAM	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**
cost	207	222	222	242	253	287	297	379	384	401
flash	400	454	466	562	436	592	492	510	646	522
RAM	64	108	108	168	176	312	324	544	552	576
cost	**172**	**191**	**191**	**209**	**221**	**253**	**264**	**337**	**339**	**357**

B Comparison Data and Methodology

Fair comparisons adhere to a common framework. The framework provides three important pieces of information. First, it provides a high-level, device-independent description of what the cipher is expected to do. At a lower level, device-dependent implementation details are provided. Finally, a performance metric is chosen to make the comparisons meaningful. Needless to say, any ranking depends on the framework used, especially the performance metric. The following application types are especially relevant to lightweight cryptography.

- **Fixed Key/Small Data.** Fixed key applications assume the key will never (or rarely) be changed. In hardware, area requirements can be reduced (depending on the design and the implementation) because state for a key schedule may not be required. In software, the key, or better yet the expanded key, can be stored in long-term memory and the key schedule can be relinquished. For small data size comparisons, we may assume we are encrypting just a single block and so incur the full cost of any setup. This application type may be appropriate for simple authentication applications.
- **Fixed Key/Large Data.** This is the same as the previous type except that the data stream is assumed to be large. For comparison purposes, we may assume the amount of encrypted data approaches infinity, amortizing away the setup costs. This may be appropriate for various sensor applications.
- **Flexible Key/Small Data.** Here, we assume the key is changed often. In hardware or software, this necessitates the inclusion of the key schedule. For comparison purposes, we may assume that we are encrypting a single block of data with a never-before-seen key. This type of application may be appropriate when the block cipher is contained in a general purpose crypto module and the key enters the device from outside of the module.
- **Flexible Key/Large Data.** Similar to the preceeding except the data stream is large. For comparison purposes, we may again assume the data stream is (effectively) infinite, amortizing away all setup costs.

It is generally recognized in lightweight cryptography that use of the decryption operator should be avoided, if possible, in order to conserve resources. If resources are not a big concern, one should use AES. For software applications on a microcontroller, implementations should be assembly coded in order to reduce compiler vagaries and to provide for maximal performance (i.e., to reduce code size and memory usage and to increase throughput).

For our comparisons, shown in Table 9, we (mostly) used the Fixed Key/ Small Data framework. For our implementations, expanded key is stored in flash and only encryption functionality is provided. Key schedules are also absent. The encryption procedure begins by loading the plaintext from RAM into registers. The plaintext is then transformed into the ciphertext using the encryption operator. The resulting ciphertext is then loaded in RAM. This completes the encryption process. RAM for holding the plaintext and ciphertext is not costed

Table 9. Comparisons of SIMON and SPECK with some other block and stream ciphers on the ATmega128, 8-bit microcontroller in the Fixed Key/Small Data framework. Values in square brackets, [ ], are our best estimates based on the existing literature. The higher the rank, the better the *overall* performance. Since our goal was to optimize the rank, these implementations are not necessarily the fastest possible.

Size	Name	Flash (bytes)	RAM (bytes)	Cost (bytes)	Rank
Comparisons with Block Ciphers					
48/96	SPECK	158	0	140	45.2
	SIMON	190	0	222	23.7
64/80	TWINE	1208	23	326	2.4
	KLEIN	766	18	762	1.6
80/80	ITUBEE	586	0	294	5.8
64/96	SPECK	214	0	148	31.6
	SIMON	282	0	242	14.7
	KLEIN	[766.]	[18.]	[955.]	[1.3]
64/128	SPECK	218	0	154	29.8
	SIMON	290	0	253	13.6
	HIGHT	336	0	311	9.6
	IDEA	[397.]	0	338	[7.5]
	TWINE	1208	23	326	2.4
	TEA	350	12	638	4.2
96/96	SPECK	324	0	152	20.3
	SIMON	474	0	287	7.4
	SEA	[1066.]	0	805	[1.2]
128/128	SPECK	460	0	171	12.7
	AES	970	18	146	6.8
	SIMON	760	0	379	3.5
Comparisons with Stream Ciphers					
128/256	SPECK	476	0	181	11.6
	AES	1034	18	204	4.7
	SIMON	792	0	401	3.1
512/256	Salsa 20/12	1092	275	177	3.4
384/128	Sosemanuk	11140	712	118	0.7
32768/128	HC-128	23100	4556	169	0.2

but RAM used for temporary storage (e.g., on the stack) is. Our low-RAM implementations of SIMON and SPECK were appropriate for this comparison. For the other ciphers, implementations fitting the framework were based on the best existing code (or performance data) we could find which maximized the overall performance metric, *rank*, which is defined to be

$$(10^6/\text{cost})/(\text{flash} + 2 \cdot \text{RAM});$$

higher values of rank correspond to better performance.[7] In some cases, this just amounted to stripping out the code for the decryption and key schedule algorithms in existing implementations. In other cases, we wrote the code ourselves.

Note that our performance metric is an *overall* measure of performance and takes into account flash, RAM and throughput. However, depending on priorities, this metric may be irrelevant. If the main concern is energy efficiency, then a more appropriate metric is just throughput and a fair comparison will require implementations optimized for this purpose, resulting in altered rankings. We have already alluded to this in our discussion in Sect. 6.

Referring to Table 9, all implementations, except for HC-128, were assembly coded. *Size* is block size/key size for block ciphers and state size/key size for stream ciphers. The *cost* is the number of cycles per byte to transform a block of plaintext into a block of ciphertext.

The ITUBEE data is taken directly from its specification paper [16]. Data for SEA, IDEA and KLEIN are taken from [10,11,18], respectively. In some cases, code size estimates had to be made to fit our framework. The TEA and HIGHT implementations are our own. For AES, our numbers were kindly provided by Dag Arne Osvik, one of the authors of [8], who made suitable code modifications to fit our framework. The TWINE data, fitting our framework, was provided by two of the TWINE designers, Kazuhiko Minematsu and Tomoyasu Suzaki. Our numbers for Salsa 20/12 were obtained by scaling down the cost of the Salsa 20/20 implementation provided in [15]. Data for Sosemanuk and HC-128 was obtained from [17]. We did not include the considerable setup time for HC-128 and the moderate setup time for Sosemanuk, and of course this setup time *should* be considered in the Fixed Key/Small Data framework.[8]

References

1. Atmel Corporation. 8-bit AVR Instruction Set, Rev. 0856I-AVR-07/10. http://www.atmel.com/images/doc0856.pdf
2. Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., Wingers, L. The Simon and Speck Families of Lightweight Block Ciphers. Cryptology ePrint Archive, Report 2013/404 (2013). http://eprint.iacr.org/
3. Berbain, C., Billet, O., Canteaut, A., Courtois, N., Gilbert, H., Goubin, L., Gouget, A., Granboulan, L., Lauradoux, C., Minier, M., Pornin, T., Sibert, H. SOSEMANUK, a fast software-oriented stream cipher. In: CoRR, abs/0810.1858 (2008)
4. Bernstein, D.: 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)

[7] The *rank* is similar to the metric found in [21] except we have imposed a penalty for using too much RAM — hence the factor of 2. Without the factor of 2, flash and RAM have the same cost, which seems unjustifiable.

[8] The HC-128 stream cipher implementation does not actually fit on the ATmega128 due to its excessive use of RAM. The C implementation of HC-128 described in [17] has a setup cost of over 2,000,000 cycles.

5. Boesgaard, M., Vesterager, M., Pedersen, T., Christiansen, J., Scavenius, O.: Rabbit: a new high-performance stream cipher. In: Johansson, T. (ed.) Fast Software Encryption, vol. 2887, pp. 307–329. Springer, Heidelberg (2003)
6. Bogdanov, A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J.B., Seurin, Y., Vikkelsoe, C.: PRESENT: an ultra-lightweight block cipher. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 450–466. Springer, Heidelberg (2007)
7. Borghoff, J., Canteaut, A., Güneysu, T., Kavun, E.B., Knežević, M., Knudsen, L.R., Leander, G., Nikov, V., Parr, C., Rechberger, C., Rombouts, P., Thomsen, S.S., Yalçın, T.: PRINCE – a low-latency block cipher for pervasive computing applications. In: Wang, X., Sako, K. (eds.) ASIACRYPT 2012. LNCS, vol. 7658, pp. 208–225. Springer, Heidelberg (2012)
8. Bos, J., Osvik, D., Stefan, D.: Fast Implementations of AES on Various Platforms. Cryptology ePrint Archive, Report 2009/501 (2009). http://eprint.iacr.org/
9. 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)
10. Eisenbarth, T., Gong, Z., Güneysu, T., Heyse, S., Indesteege, S., Kerckhof, S., Koeune, F., Nad, T., Plos, T., Regazzoni, F., Standaert, F., van Oldeneel tot Oldenzeel, L.: Compact implementation and performance evaluation of block ciphers in attiny devices. In: Mitrokotsa, A., Vaudenay, S. (eds.) AFRICACRYPT 2012. LNCS, vol. 7374, pp. 172–187. Springer, Heidelberg (2012)
11. Eisenbarth, T., Kumar, S., Paar, C., Poschmann, A., Uhsadel, L.: A survey of lightweight cryptography implementations. IEEE Des. Test Comput. **24**(6), 522–533 (2007)
12. Gong, Z., Nikova, S., Law, Y.W.: KLEIN: a new family of lightweight block ciphers. In: Juels, A., Paar, C. (eds.) RFID. Security and Privacy. LNCS, vol. 7055, pp. 1–18. Springer, Heidelberg (2011)
13. Guo, J., Peyrin, T., Poschmann, A., Robshaw, M.: The LED block cipher. In: Preneel, B., Takagi, T. (eds.) CHES 2011. LNCS, vol. 6917. Springer, Heidelberg (2011)
14. Hong, D., Sung, J., Hong, S., Lim, J., Lee, S., Koo, B., Lee, C., Chang, D., Lee, J., Jeong, K., Kim, H., Kim, J., Chee, S.: HIGHT: A new block cipher suitable for low-resource device. In: Goubin, L., Matsui, M. (eds.) CHES 2006. LNCS, vol. 4249, pp. 45–59. Springer, Heidelberg (2006)
15. Hutter, M., Schwabe, P.: NaCl on 8-bit AVR microcontrollers. In: Youssef, A., Nitaj, A., Hassanien, A.E. (eds.) AFRICACRYPT 2013. LNCS, vol. 7918, pp. 156–172. Springer, Heidelberg (2013)
16. Karakoç, F., Demirci, H., Emre Harmancı, A.: ITUBEE: a software oriented lightweight block cipher. In: Avoine, G., Kara, O. (eds.) Lightweight Cryptography for Security and Privacy. LNCS, vol. 8162, pp. 16–27. Springer, Heidelberg (2013)
17. Meiser, G.: Efficient implementation of stream ciphers on embedded processors. Masters Thesis, Ruhr-University Bochum (2007)
18. Rinne, S., Eisenbarth, T., Paar, C.: Performance analysis of contemporary lightweight block ciphers on 8-bit microcontrollers. In: SPEED - Software Performance Enhancement for Encryption and Decryption (2007)
19. Shibutani, K., Isobe, T., Hiwatari, H., Mitsuda, A., Akishita, T., Shirai, T.: Piccolo: an ultra-lightweight blockcipher. In: Preneel, B., Takagi, T. (eds.) CHES 2011. LNCS, vol. 6917, pp. 342–357. Springer, Heidelberg (2011)

20. Shirai, T., Shibutani, K., Akishita, T., Moriai, S., Iwata, T.: The 128-bit blockcipher CLEFIA (extended abstract). In: Biryukov, A. (ed.) Fast Software Encryption. LNCS, vol. 4593, pp. 181–195. Springer, Heidelberg (2007)
21. Suzaki, T., Minematsu, K., Morioka, S., Kobayashi, E.: TWINE: a lightweight, versatile block cipher. www.nec.co.jp/rd/media/code/research/images/twine_LC11.pdf
22. Wei, L., Rechberger, C., Guo, J., Wu, H., Wang, H., Ling, S.: Improved meet-in-the-middle cryptanalysis of KTANTAN. Inf. Secur. Priv. ACISP **2011**, 433–438 (2011)
23. Wheeler, D., Needham, R.: TEA, a tiny encryption algorithm. In: Preneel, B. (ed.) Fast Software Encryption. LNCS, vol. 1008, pp. 363–366. Springer, Heidelberg (1995)
24. Wu, H.: The stream cipher HC-128. www.ecrypt.eu.org/stream/p3ciphers/hc/hc128_p3.pdf
25. Wu, W., Zhang, L.: LBlock: a lightweight block cipher. In: Lopez, J., Tsudik, G. (eds.) Applied Cryptography and Network Security. LNCS, vol. 6715, pp. 327–327. Springer, Heidelberg (2011)

The Multiplicative Complexity of Boolean Functions on Four and Five Variables

Meltem Turan Sönmez[1,2](✉) and René Peralta[1]

[1] National Institute of Standards and Technology, Gaithersburg, MD, USA
{meltem.turan,rene.peralta}@nist.gov

[2] Dakota Consulting Inc., Silver Spring, MD, USA

Abstract. A generic way to design lightweight cryptographic primitives is to construct simple rounds using small nonlinear components such as 4×4 S-boxes and use these iteratively (e.g., PRESENT [1] and SPONGENT [2]). In order to efficiently implement the primitive, efficient implementations of its internal components are needed. Multiplicative complexity of a function is the minimum number of AND gates required to implement it by a circuit over the basis (AND, XOR, NOT). It is known that multiplicative complexity is exponential in the number of input bits n. Thus it came as a surprise that circuits for all 65 536 functions on four bits were found which used at most three AND gates [3]. In this paper, we verify this result and extend it to five-variable Boolean functions. We show that the multiplicative complexity of a Boolean function with five variables is at most four.

Keywords: Affine transformation · Boolean functions · Circuit complexity · Multiplicative complexity

1 Introduction

One of the important challenges in lightweight cryptography is to find efficient implementations of secure cryptographic primitives. Many attempts [4–7] have been done to improve the efficiency of the block cipher AES, in order to fit the implementations in resource-constrained devices. However, even the best implementations of AES are usually too big for constrained devices. A generic way to design dedicated lightweight cryptographic primitives is to construct simple rounds using small nonlinear components such as 4×4 S-boxes and iterate (this is done in, for example, PRESENT [1] and SPONGENT [2]). In order to efficiently implement the primitive, efficient circuits for the internal components are needed. Efficiency of the implementations can be assessed using different metrics such as area, power, and energy requirements. These metrics are strongly related to the number of logic gates used to implement the primitive.

The rights of this work are transferred to the extent transferable according to title 17 § 105 U.S.C.

© Springer International Publishing Switzerland 2015 (outside the US)
T. Eisenbarth and E. Öztürk (Eds.): LightSec 2014, LNCS 8898, pp. 21–33, 2015.
DOI: 10.1007/978-3-319-16363-5_2

Gate complexity is defined as the minimum number of 2-input logic gates required to implement the primitive in a circuit. *Multiplicative complexity* (MC) is another complexity measure, which is defined as the minimum number of AND gates required to implement the primitive by a circuit over the basis (AND, XOR, NOT), with an unlimited number of NOT and XOR gates. This is the same as the number of multiplications needed for straight-line programs that do arithmetic modulo 2.

Finding the gate complexity or the multiplicative complexity of a given function is computationally intractable, even for functions with a small number of variables. In 2006, Saarinen [8] published the gate complexity distribution of 4-variable Boolean functions. In 2010, Boyar et al. [9] proposed a two-stage heuristic method to minimize the gate complexity of Boolean circuits. In the first state, the heuristics minimizes the number of AND gates required to implement the circuit, and then in the second stage, the linear components are optimized. Using this method, they constructed efficient circuits for the AES S-box over the basis (AND, XOR, NOT).

Apart from possibly increasing the efficiency of the implementations, minimizing the number of AND gates provides a tool for the cryptanalysis of the primitives. For example, according to [10,11], functions with low multiplicative complexity are more vulnerable to algebraic attacks than those with high multiplicative complexity. This is an important observation, as a function representation tends to hide its true multiplicative complexity (e.g. consider a random polynomial on five variables over $GF(2)$, it is hard to see how it could possibly be computed using only four multiplications).

Also, a relationship between collision resistance of a hash function and multiplicative complexity is provided in [12]. Courtois et al. [10] argued that minimizing the number of AND gates is important to prevent against side channel attacks such as differential power analysis. Finally, we point out that the number and position of AND gates in a circuit is the main determinant of whether the function can be used in the context of protocols that use homomorphic encryption.

Multiplicative complexity of a randomly selected n-variable Boolean function is at least $2^{n/2} - O(n)$ [13]. Exhaustive study of the distribution of multiplicative complexity can only be done for very small values of n. In 2013, we were surprised to find circuits for all 65 536 functions on four bits which used at most three AND gates [3]. Boyar et al. [12] conjecture that some five-bit Boolean functions have multiplicative complexity five. It is shown in [13] that there are at most $2^{k^2+2k+2kn+n+1}$ many n-variable Boolean functions that can be generated using k AND gates. This is a counting argument and it is not known how tight it is. Using this bound, it is easy to see that there exist eight-bit Boolean functions with multiplicative complexity of at least eight. So, it is an open question whether there exists n-bit Boolean functions with multiplicative complexity n, for $n = 5, 6, 7$. Lower bounds are extremely difficult: no specific n-variable function has yet been proven to have multiplicative complexity larger than $n-1$ for any n.

In this paper, we focus on the multiplicative complexity of four and five-variable Boolean functions. Using the fact that multiplicative complexity is affine

invariant, we first provide a succinct proof (i.e. one that does not list all circuits) that the multiplicative complexity of four-variable Boolean functions is at most three. Then, we extend the result for five-variable Boolean functions and show that the conjecture given in [12] is false: any five-bit Boolean functions can be implemented with at most four AND gates.

The organization of this paper is as follows. Section 2 provides definitions and some known facts about Boolean functions, affine equivalence, and multiplicative complexity. Section 2 focuses on affine invariance of multiplicative complexity and provides results for four and five-variable Boolean functions. Section 4 concludes the paper.

2 Preliminaries

2.1 Boolean Functions

Let $\mathbb{F}_2$ be the binary field. An n-variable Boolean function f is a mapping from $\mathbb{F}_2^n$ to $\mathbb{F}_2$. Let B_n be the set of n-variable Boolean functions. Note that $|B_n| = 2^{2^n}$. Boolean functions have various representations, some are canonical, some not so. The list of output values for each n-bit input $T_f = (f(0,\ldots,0), f(0,\ldots,0,1),\ldots, f(1,\ldots,1))$ is called the *truth table* representation of f. The number of ones in T_f is called the *Hamming weight* of f, denoted $wt(f)$. The *Hamming distance* between two Boolean functions $f, g \in B_n$, denoted $d(f,g)$, is $wt(f+g)$. That is, the Hamming distance is the cardinality of the set $\{x \in \mathbb{F}_2^n | f(x) \neq g(x)\}$.

Boolean functions are also represented uniquely by the multivariate polynomial called *algebraic normal form* (ANF)

$$f(x_1,\ldots,x_n) = \sum_{u \in \mathbb{F}_2^n} a_u x^u \quad (1)$$

where $x^u = x_1^{u_1} x_2^{u_2} \cdots x_n^{u_n}$ is a *monomial* composed of the variables for which $u_i = 1$ and $a_u \in \mathbb{F}_2$. This is also called the *Zhegalkin polynomial* of f. The degree of a Boolean function, denoted d_f, is the degree of the highest-degree monomial in its ANF representation.

Let A_n be the set of n-variable affine functions, i.e., functions having degree at most one. The nonlinearity of a Boolean function f, denoted N_f, is the minimum Hamming distance of f to all affine functions, that is $N_f = \min_{g \in A_n} d(f,g)$. The nonlinearity of an n-variable Boolean function is upper bounded by $2^{n-1} - 2^{n/2-1}$.

2.2 Affine Equivalence

Affine transformations of a function f allow for linear operations to inputs and output of f. There are several definitions in the literature, and they are not all equivalent to each other. Here, we will use a definition from Berlekamp and Welch [14].

Definition 1. *An* affine transformation *from g to f in B_n is a mapping of the form $f(x) = g(Ax + a) + b \cdot x + c$, where*

- *A is a non-singular $n \times n$ matrix over $\mathbb{F}_2$;*
- *x, a are column vectors over $\mathbb{F}_2$;*
- *b is a row vector over $\mathbb{F}_2$; and*
- *$c \in F_2$.*

It is not hard to prove that this defines an equivalence relation on the set of n-variable Boolean functions. Two functions f, g are *affine equivalent* if there exist affine transformations between them. Affine equivalent Boolean functions are said to be in the same equivalence class.

An algorithm to check whether two functions are equivalent is given in [15]. This algorithm also outputs an affine transformation between the input functions, if one exists. The equivalence classes can be constructed using exhaustive search for small values of n. For example, the Boolean functions with three variables can be partitioned into three equivalence classes, and a representative from each class can be given as $\{x_1, x_1x_2$ and $x_1x_2x_3\}$. The classification of five-variable Boolean functions was done in 1972 by Berlekamp and Welch [14]. Maiorana [16] proved that the number of classes in B_6 is 150 357. This was independently verified by Fuller [15] and by Braeken et al. [17]. For $n = 7$, Hou [18] determined that there are 63 379 147 320 777 408 548 equivalence classes. Since the size of B_n is doubly exponential, and each equivalence class can contain only an exponential number of functions, the number of equivalence classes is asymptotically exponential in n (see Table 1).

Table 1. Number of equivalence classes for $n \leq 7$

n	$\|B_n\|$	# of equivalence classes
3	2^8	3
4	2^{16}	8
5	2^{32}	48
6	2^{64}	150 357
7	2^{128}	63 379 147 320 777 408 548 (this is between 2^{65} and 2^{66})

Some cryptographic measures such as nonlinearity, algebraic degree, and algebraic immunity remain unchanged after applying an affine transformation. Such measures are said to be *affine invariant* [19]. Braeken et al. [17] studied the classification of Boolean functions with respect to various cryptographic properties. Uyan [20] analyzed the Boolean functions with respect to the Walsh Spectrum using equivalence classes.

2.3 Multiplicative Complexity

The *multiplicative complexity* $C_\wedge(f)$ of a Boolean function is the minimum number of multiplications (AND-$\wedge$ gates) that are sufficient to evaluate the function

over the basis (AND, XOR, NOT). The multiplicative complexity of functions having degree d is at least $d-1$ [21]. This bound is called the *degree bound*. Calculating the multiplicative complexity of a randomly selected Boolean function is hard even for small values of n[1].

3 Multiplicative Complexity of Boolean Functions

Multiplicative complexity is invariant under affine transformations. Thus, to bound the multiplicative complexity distribution of n-bit Boolean functions, it is enough to bound the multiplicative complexity of a single function from each equivalence class. Fuller presents an algorithm to find a representative from each equivalence class [15]. Her method is practical for values of n up to six.

In order to find a circuit for $f \in B_n$ with a small number of AND gates, we use the following approach.

1. Precomputation
   (a) Using the algorithm given in [15], find a "simple" (e.g., one that has a small number of monomials in its ANF) representative for each equivalence class in B_n.
   (b) For each representative, find a circuit which is efficient with respect to multiplicative complexity.
2. Find the equivalence class C_f of f.
3. Find the affine transformation from f^*, the representative of C_f, to f using the algorithm given in [15].
4. Apply the affine transformation to the circuit for f^*. This yields a circuit for f. The circuit will be efficient with respect to multiplicative complexity. Note that, if the circuit found for f^* is not optimal, this still yields an upper bound on the multiplicative complexity of f.

In the following subsections, we used this approach to bound the multiplicative complexity of all Boolean functions on four and on five variables. The approach becomes impractical as the number of variables increases due to the following reasons; (i) the number of equivalence classes increases exponentially with the number of variables; (ii) finding an affine transformation from f^* to f gets harder; and (iii) constructing circuits that are optimal with respect to multiplicative complexity gets harder.

3.1 $n = 4$

There are eight equivalence classes of B_4, with representatives $\{x_1,\ x_1x_2,\ x_1x_2 + x_3x_4,\ x_1x_2x_3,\ x_1x_2x_3 + x_1x_4,\ x_1x_2x_3x_4,\ x_1x_2x_3x_4 + x_1x_2,\ x_1x_2x_3x_4 + x_1x_2 + x_3x_4\}$. The representatives are simple enough that optimal (with respect to

[1] Lest the reader think this easy, he/she may attempt to compute the function $f(x_1, x_2, x_3, x_4, x_5) = x_1x_2x_3x_4x_5 + x_1x_2x_3 + x_1x_2x_4 + x_2x_3x_4 + x_1x_2 + x_1x_3 + x_1x_4 + x_2x_4 + x_3x_4$ using only four AND gates.

Table 2. Equivalence classes of B_4

Class	Representatives	Implementation	MC	(N_f, d_f)	# Functions
1	x_1	$f = x_1$	0	(0,1)	32
2	x_1x_2	$t_1 = x_1 \wedge x_2$	1	(4,2)	1120
3	$x_1x_2 + x_3x_4$	$t_1 = x_1 \wedge x_2$ $t_2 = x_3 \wedge x_4$ $f = t_1 \oplus t_2$	2	(6,2)	896
4	$x_1x_2x_3$	$t_1 = x_1 \wedge x_2$ $f = t_1 \wedge x_3$	2	(2,3)	3840
5	$x_1x_2x_3 + x_1x_4$	$t_1 = x_2 \wedge x_3$ $t_2 = t_1 \oplus x_4$ $f = t_2 \wedge x_1$	2	(4,3)	26880
6	$x_1x_2x_3x_4$	$t_1 = x_1 \wedge x_2$ $t_2 = t_1 \wedge x_3$ $f = t_2 \wedge x_4$	3	(1,4)	512
7	$x_1x_2x_3x_4 + x_1x_2$	$t_1 = x_1 \wedge x_2$ $t_2 = t_1 \wedge x_3$ $t_3 = t_2 \wedge x_4$ $f = t_3 \oplus t_1$	3	(3,4)	17920
8	$x_1x_2x_3x_4 + x_1x_2 + x_3x_4$	$t_1 = x_1 \wedge x_2$ $t_2 = x_3 \wedge x_4$ $t_3 = t_1 \wedge t_2$ $t_4 = t_3 \oplus t_1$ $f = t_4 \oplus t_2$	3	(5,4)	14336

multiplicative complexity) circuits can be easily constructed. Table 2 provides a circuit with optimal number of AND gates for each of these representatives. The optimality of the circuits follows from the degree bound for seven (out of eight) of the equivalence classes. For example, according to the degree bound, the multiplicative complexity of $x_1x_2x_3x_4 + x_1x_2$, is at least three. Optimality of the third class, which is a sum of quadratic functions, cannot be verified by the degree bound. That two AND gates are needed seems obvious, as the two quadratic terms have no common variables. For a formal proof, see Mirwald and Schnorr [22], which shows that the multiplicative complexity of a quadratic function of the form $\sum_{i=1}^{k} x_{2i-1}x_{2i}$ is k.

It is easy to find the equivalence class of a given function $f \in B_4$ by checking the nonlinearity and degree of f, since the nonlinearity and the degree pair (N_f, d_f) of the representatives are distinct (See Table 2).

Example 1. Let $f = 1 + x_1 + x_2 + x_3 + x_1x_2 + x_1x_3 + x_2x_4 + x_3x_4 + x_1x_2x_3 + x_2x_3x_4 + x_1x_3x_4 + x_1x_2x_3x_4$. In order to find a circuit for f with minimum

number of AND gates, we first need to find its equivalence class. Since (N_f, degree) = (3,4), f belongs to the seventh equivalence class with representative $f^* = x_1x_2x_3x_4 + x_1x_2$. Then, we need to obtain the affine transformation between f and f^*. Using the algorithm in [15], the transformation is obtained as

$$f(x) = f^*(\begin{pmatrix} 1\,0\,1\,1 \\ 1\,1\,1\,1 \\ 0\,1\,1\,0 \\ 1\,0\,1\,0 \end{pmatrix} x \oplus (0\ 0\ 0\ 0)) \oplus (0\ 1\ 1\ 0)x \oplus 1. \qquad (2)$$

According to this transformation, input variables are transformed as follows; $x_1 \rightarrow x_1 + x_3 + x_4$, $x_2 \rightarrow x_1 + x_2 + x_3 + x_4$, $x_3 \rightarrow x_2 + x_3$, and $x_4 \rightarrow x_1 + x_3$, and the affine shift is equivalent to XORing $x_2 + x_3 + 1$ to f. Given an efficient circuit for $x_1x_2x_3x_4 + x_1x_2$, an efficient circuit for f can be found as is shown in Table 3. The first four equations in the implementation of f are due to the linear transformations of input variables, whereas the last three equations corresponds to the affine shift.

Table 3. Optimal circuit for f in terms of number of AND gates.

f^*	f
	$t_1 = x_1 \oplus x_3$
	$t_2 = t_1 \oplus x_4$
	$t_3 = t_2 \oplus x_2$
	$t_4 = x_2 \oplus x_3$
$t_1 = x_1 \wedge x_2$	$t_5 = t_2 \wedge t_3$
$t_2 = t_1 \wedge x_3$	$t_6 = t_5 \wedge t_4$
$t_3 = t_2 \wedge x_4$	$t_7 = t_6 \wedge x_3$
$f* = t_3 \oplus t_1$	$t_8 = t_7 \oplus t_5$
	$t_9 = t_8 \oplus t_3$
	$t_{10} = t_9 \oplus t_4$
	$f = t_{10} \oplus 1$

□

3.2 $n = 5$

Berlekamp and Welsh [14] provided the representatives of the 48 equivalence classes for $n = 5$. Table 5 in the Appendix provides the circuits to implement the representatives of each equivalence class. Most of the representatives are simple enough that the optimal circuits are found trivially. The circuits corresponding to the representatives of the classes 14, 18, 26, 37, 44, 45, 46, 47, 48 are obtained using the heuristic provided in [23]. Optimality of thirty circuits (out of 48) can be verified using the degree bound.

To find the equivalence class of a given function $f \in B_5$, the nonlinearity and degree of f can be utilized. Table 4 provides a classification of the equivalence classes based on the nonlinearity of degrees. As seen from the table, for 10 of the equivalence classes, the degree and nonlinearity uniquely determines the class representative. Moreover, checking degree and nonlinearity of an input function significantly reduces the possible number of equivalence classes.

Table 4. The distribution of equivalence classes of B_5 according to degree and nonlinearity. The functions are written in an abbreviated notation. For example, 123+145 indicates the representative of the form $x_1x_2x_3 + x_1x_4x_5$.

	d_f				
N_f	1	2	3	4	5
0	1	-	-	-	-
1	-	-	-	-	12345
2	-	-	-	2345	-
3	-	-	-	-	12345+123
4	-	-	123	2345+123	-
5	-	-	-	-	12345+123+12 12345+123+145+12
6	-	-	123+145	2345+23 2345+123+12 2345+123+145+45	-
7	-	-	-	-	12345+12 12345+123+14 12345+123+145 12345+123+145+23 12345+123+145+45+23
8	-	12	123+14 123+145+23	2345+12 2345+123+24 2345+123+14 2345+123+145 2345+123+12+45	-
9	-	-	-	-	12345+123+45 12345+123+12+45 12345+123+12+34 12345+123+145+24 12345+123+145+24+23 12345+123+145+35+24
10	-	-	123+45 123+145+24	2345+23+45 2345+12+34 2345+123+45 2345+123+12+34 2345+123+14+35 2345+123+145+24+45 2345+123+145+35+24	-
11	-	-	-	-	12345+12+34 12345+123+14+25 12345+123+145+35+24+23 12345+123+145+45+35+24+23
12	-	12+34	123+14+25 123+145+23+24+35	2345+123+24+35	-

4 Conclusion

We studied the multiplicative complexity of Boolean functions with four and five variables. For four variables, we confirmed that the multiplicative complexity is at most three by producing circuits for a representative of each of the eight equivalence classes. We knew this was true because one of us has posted circuits for all 2^{16}, each using at most three AND gates [3]. Those circuits are also optimized for total number of gates: it turns out that no more than seven XOR gates are needed by AND-optimal circuits.

For five variables, we disproved the conjecture that there exists Boolean functions with multiplicative complexity five. We are in the process of extending this work to six-variable Boolean functions. This will most likely require a computer proof, as there are 150 357 equivalence classes.

Acknowledgments. We thank Çağdaş Çalık, Joan Boyar, and Magnus Find for helpful discussions and suggestions. We also thank our colleagues Yi-Kai Liu, Ray Perlner, Lily Chen, and the anonymous reviewers for their useful comments.

Appendix

Table 5. Circuits for $n = 5$.

Class	Representative	Circuit	MC
1	2345	$t_1 = 2 \wedge 3,\ t_2 = 4 \wedge 5,\ f = t_1 \wedge t_2$	3
2	2345⊕12	$t_1 = 3 \wedge 4,\ t_2 = t_1 \wedge 5,\ t_3 = t_2 \oplus 1,\ f = 2 \wedge t_3$	3
3	2345⊕23	$t_1 = 2 \wedge 3,\ t_2 = t_1 \wedge 4,\ t_3 = t_2 \wedge 5,\ f = t_1 \oplus t_3$	3
4	2345⊕23 ⊕ 45	$t_1 = 2 \wedge 3,\ t_2 = 4 \wedge 5,\ t_3 = t_1 \wedge t_2,\ t_4 = t_3 \oplus t_1$ $f = t_4 \oplus t_2$	3
5	2345⊕12⊕34	$t_1 = 3 \wedge 4,\ t_2 = t_1 \wedge 5,\ t_3 = t_2 \oplus 1,\ t_4 = 2 \wedge t_3,$ $f = t_4 \oplus t_1$	3
6	2345⊕123	$t_1 = 2 \wedge 3,\ t_2 = 4 \wedge 5,\ t_3 = t_2 \oplus 1,\ f = t_3 \wedge t_1$	3
7	2345⊕123⊕12	$t_1 = 4 \wedge 5,\ t_2 = 1 \oplus t_1,\ t_3 = 3 \wedge t_2,\ t_4 = 1 \oplus t_3$ $f = 2 \wedge t_4$	3
8	2345⊕123⊕24	$t_1 = 4 \wedge 5,\ t_2 = 1 \oplus t_1,\ t_3 = 3 \wedge t_2,\ t_4 = 4 \oplus t_3$ $f = 2 \wedge t_4$	3
9	2345⊕123⊕14	$t_1 = 2 \wedge 3,\ t_2 = 4 \wedge 5,\ t_3 = 1 \oplus t_2,\ t_4 = t_1 \wedge t_3$ $f = t_4 \oplus t_2$	3
10	2345⊕123⊕45	$t_1 = 2 \wedge 3,\ t_2 = 4 \wedge 5,\ t_3 = t_2 \oplus 1,\ t_4 = t_3 \wedge t_1,$ $f = t_4 \oplus t_2$	3
11	2345⊕123⊕12⊕34	$t_1 = 2 \wedge 4,\ t_2 = t_1 \wedge 5,\ t_3 = 1 \wedge 2,\ t_4 = t_2 \oplus t_3,$ $t_5 = t_4 \oplus 4,\ t_6 = t_5 \wedge 3,\ f = t_6 \oplus t_3$	≤ 4
12	2345⊕123⊕14⊕35	$t_1 = 4 \wedge 5,\ t_2 = 1 \oplus t_1,\ t_3 = 2 \wedge t_2,\ t_4 = 5 \oplus t_3,$ $t_5 = 3 \wedge t_4,\ t_6 = 1 \wedge 4,\ f = t_5 \oplus t_6$	≤ 4
13	2345⊕123⊕12⊕45	$t_1 = 2 \wedge 3,\ t_2 = 1 \wedge 2,\ t_3 = 4 \wedge 5,\ t_4 = t_2 \oplus t_3$ $t_5 = t_1 \wedge t_4,\ f = t_5 \oplus t_4$	≤ 4
14	2345⊕123⊕24⊕35	$t_1 = 4 \wedge 5,\ t_2 = 1 \oplus t_1,\ t_3 = 2 \oplus 3,\ t_4 = 1 \oplus t_3,$ $t_5 = t_4 \oplus t_1,\ t_6 = t_2 \wedge t_5,\ t_7 = 4 \oplus t_6,\ t_8 = 2 \wedge t_7,$ $t_9 = 3 \wedge 5,\ f = t_8 \oplus t_9$	≤ 4

(*Continued*)

Table 5. (*Continued*)

Class	Representative	Circuit	MC
15	2345⊕123⊕145	$t_1 = 2 \wedge 3,\ t_2 = 1 \oplus t_1,\ t_3 = 4 \wedge 5\ t_4 = 1 \oplus t_3$ $t_5 = t_2 \wedge t_4,\ f = 1 \oplus t_5$	3
16	2345⊕123⊕145⊕45	$t_1 = 2 \wedge 3,\ t_2 = 1 \oplus t_1,\ t_3 = 4 \wedge 5,\ t_4 = 1 \oplus t_3$ $t_5 = t_2 \wedge t_4,\ f = t_5 \oplus t_4$	3
17	2345⊕123⊕145⊕24⊕45	$t_1 = 4 \wedge 5,\ t_2 = 1 \oplus t_1,\ t_3 = 2 \wedge 3,\ t_4 = t_3 \oplus t_1,$ $t_5 = t_2 \wedge t_4,\ t_6 = 2 \wedge 4,\ f = t_5 \oplus t_6$	≤ 4
18	2345⊕123⊕145⊕35⊕24	$t_1 = 2 \wedge 3,\ t_2 = 1 \oplus t_1,\ t_3 = t_4 \wedge t_5,\ t_4 = t_1 \oplus t_3,$ $t_5 = t_2 \wedge t_4,\ t_6 = 2 \oplus 5,\ t_7 = 3 \oplus 4,\ t_8 = t_6 \wedge t_7,$ $t_9 = t_3 \oplus t_8,\ f = t_5 \oplus t_9$	≤ 4
19	123	$t_1 = 1 \wedge 2,\ f = t_1 \wedge 3$	2
20	123⊕45	$t_1 = 1 \wedge 2,\ t_2 = t_1 \wedge 3,\ t_3 = 4 \wedge 5,\ f = t_2 \oplus t_3$	3
21	123⊕14	$t_1 = 2 \wedge 3,\ t_2 = t_1 \oplus 4,\ f = 1 \wedge t_2$	2
22	123⊕14⊕25	$t_1 = 2 \wedge 3,\ t_2 = t_1 \oplus 4,\ t_3 = t_2 \wedge 1,\ t_4 = 2 \wedge 5$ $f = t_3 \oplus t_4$	3
23	123⊕145	$t_1 = 2 \wedge 3,\ t_2 = 4 \wedge 5,\ t_3 = t_1 \oplus t_2,\ f = 1 \wedge t_3$	3
24	123⊕145⊕23	$t_1 = 2 \wedge 3,\ t_2 = 4 \wedge 5,\ t_3 = t_1 \oplus t_2,\ t_4 = 1 \wedge t_3$ $f = t_4 \oplus t_1$	3
25	123⊕145⊕24	$t_1 = 2 \wedge 3,\ t_2 = 4 \wedge 5,\ t_3 = t_1 \oplus t_2,\ t_4 = 1 \wedge t_3$ $t_5 = 2 \wedge 4,\ f = t_4 \oplus t_5$	≤ 4
26	123⊕145⊕23⊕24⊕35	$t_1 = 1 \wedge 5,\ t_2 = 2 \oplus t_1,\ t_3 = 1 \oplus 3,\ t_4 = 3 \wedge t_3$ $t_5 = 4 \oplus t_4,\ t_6 = t_2 \wedge t_5,\ t_7 = 3 \wedge 5,\ f = t_6 \oplus t_7$	≤ 4
27	12	$f = 1 \wedge 2$	1
28	12⊕34	$t_1 = 1 \wedge 2,\ t_2 = 3 \wedge 4,\ f = t_1 \oplus t_2$	2
29	1	$f = 1$	0
30	12345	$t_1 = 1 \wedge 2,\ t_2 = t_1 \wedge 3,\ t_3 = t_2 \wedge 4,\ f = t_3 \wedge 5$	4
31	12345⊕12	$t_1 = 1 \wedge 2,\ t_2 = t_1 \wedge 3,\ t_3 = t_2 \wedge 4,\ t_4 = t_3 \wedge 5$ $f = t_4 \oplus t_1$	4
32	12345⊕12⊕34	$t_1 = 1 \wedge 2,\ t_2 = 3 \wedge 4,\ t_3 = t_1 \wedge t_2,\ t_4 = t_3 \wedge 5$ $t_5 = t_4 \oplus t_1,\ f = t_5 \oplus t_2$	4
33	12345⊕123	$t_1 = 1 \wedge 2,\ t_2 = t_1 \wedge 3,\ t_3 = t_2 \wedge 4,\ t_4 = t_3 \wedge 5$ $f = t_4 \oplus t_2$	4
34	12345⊕123⊕12	$t_1 = 1 \wedge 2,\ t_2 = t_1 \wedge 3,\ t_3 = t_2 \wedge 4,\ t_4 = t_3 \wedge 5$ $t_5 = t_4 \oplus t_1,\ f = t_5 \oplus t_2$	4
35	12345⊕123⊕14	$t_1 = 2 \wedge 3,\ t_2 = 1 \wedge 4,\ t_3 = t_2 \wedge 5,\ t_4 = t_3 \oplus 1$ $t_5 = t_1 \wedge t_4,\ f = t_2 \oplus t_5$	4
36	12345⊕123⊕45	$t_1 = 1 \wedge 2,\ t_2 = t_1 \wedge 3,\ t_3 = 4 \wedge 5,\ t_4 = t_2 \wedge t_3$ $t_5 = t_4 \oplus t_2,\ f = t_5 \oplus t_3$	4

(*Continued*)

Table 5. (*Continued*)

Class	Representative	Circuit	MC
37	12345⊕123⊕14⊕25	$t_1 = 1 \oplus 4$, $t_2 = 3 \wedge t_1$, $t_3 = 4 \oplus t_2$, $t_4 = 3 \oplus 4$ $t_5 = 2 \oplus t_4$, $t_6 = 2 \wedge 5$, $t_7 = 3 \oplus t_6$, $t_8 = t_5 \wedge t_7$ $t_9 = 1 \oplus t_8$, $t_{10} = t_3 \wedge t_9$, $f = t_{10} \oplus t_6$	4
38	12345⊕123⊕12⊕45	$t_1 = 1 \wedge 2$, $t_2 = t_1 \wedge 3$, $t_3 = 4 \wedge 5$, $t_4 = t_2 \wedge t_3$ $t_5 = t_4 \oplus t_1$, $t_6 = t_5 \oplus t_2$, $f = t_6 \oplus t_3$	4
39	12345⊕123⊕12⊕34	$t_1 = 1 \wedge 2$, $t_2 = 4 \wedge 5$, $t_3 = t_1 \wedge t_2$, $t_4 = t_3 \oplus t_1$ $t_5 = t_4 \oplus 4$, $t_6 = 3 \wedge t_5$, $f = t_1 \oplus t_6$	4
40	12345⊕123⊕145	$t_1 = 2 \wedge 3$, $t_2 = 4 \wedge 5$, $t_3 = t_1 \wedge t_2$, $t_4 = t_1 \oplus t_2$ $t_5 = t_4 \oplus t_3$, $f = 1 \wedge t_5$	4
41	12345⊕123⊕145⊕12	$t_1 = 2 \wedge 3$, $t_2 = 4 \wedge 5$, $t_3 = t_1 \wedge t_2$, $t_4 = t_1 \oplus t_2$ $t_5 = t_4 \oplus 2$, $t_6 = t_5 \oplus t_3$, $f = 1 \wedge t_6$	4
42	12345⊕123⊕145⊕23	$t_1 = 2 \wedge 3$, $t_2 = 4 \wedge 5$, $t_3 = t_1 \wedge t_2$, $t_4 = t_3 \oplus t_1$ $t_5 = t_4 \oplus t_2$, $t_6 = 1 \wedge t_5$, $f = t_6 \oplus t_1$	4
43	12345⊕123⊕145⊕45⊕23	$t_1 = 2 \wedge 3$, $t_2 = 4 \wedge 5$, $t_3 = t_1 \wedge t_2$, $t_4 = t_3 \oplus t_1$ $t_5 = t_4 \oplus t_2$, $t_6 = 1 \wedge t_5$, $t_7 = t_6 \oplus t_1$, $f = t_7 \oplus t_2$	4
44	12345⊕123⊕145⊕24	$t_1 = 2 \oplus 3$, $t_2 = 1 \oplus t_1$, $t_3 = 3 \wedge t_2$, $t_4 = 4 \oplus t_3$ $t_5 = 1 \wedge 4$, $t_6 = 5 \wedge t_5$, $t_7 = 2 \oplus t_6$, $f = t_4 \wedge t_7$	4
45	12345⊕123⊕145⊕24⊕23	$t_1 = 2 \wedge 3$, $t_2 = 3 \oplus 4$, $t_3 = 1 \oplus t_2$, $t_4 = 1 \wedge 5$ $t_5 = 2 \oplus t_4$, $t_6 = 4 \wedge t_5$, $t_7 = t_3 \oplus t_6$, $t_8 = t_1 \wedge t_7$ $f = t_8 \oplus t_6$	4
46	12345⊕123⊕145⊕35⊕24	$t_1 = 4 \oplus 5$, $t_2 = 1 \oplus t_1$, $t_3 = 5 \wedge t_2$, $t_4 = 3 \oplus t_3$ $t_5 = 2 \oplus 3$, $t_6 = 4 \wedge 5$, $t_7 = 1 \oplus t_6$, $t_8 = 2 \wedge t_7$ $t_9 = t_1 \oplus t_8$, $t_{10} = t_4 \wedge t_9$, $t_{11} = t_6 \oplus t_3$ $f = t_{10} \oplus t_{11}$	4

(*Continued*)

Table 5. (*Continued*)

Class	Representative	Circuit	MC
47	12345⊕123⊕145⊕35⊕24⊕23	$t_1 = 1 \oplus 4$, $t_2 = 1 \wedge t_1$, $t_3 = 2 \oplus t_2$, $t_4 = 4 \oplus 5$ $t_5 = 1 \oplus 3$, $t_6 = 2 \oplus 3$, $t_7 = 1 \oplus t_6$, $t_8 = 5 \wedge t_7$ $t_9 = t_5 \oplus t_8$, $t_{10} = 3 \wedge t_9$, $t_{11} = t_4 \oplus t_{10}$ $t_{12} = t_3 \wedge t_{11}$, $f = t_{12} \oplus t_8$	4
48	12345⊕123⊕145⊕45⊕35⊕24⊕23	$t_1 = 2 \oplus 4$, $t_2 = 2 \oplus 3$, $t_3 = 2 \oplus 5$, $t_4 = t_2 \wedge t_3$ $t_5 = t_1 \oplus t_4$, $t_6 = 1 \oplus t_3$, $t_7 = 4 \oplus 5$, $t_8 = t_2 \oplus t_7$ $t_9 = 1 \oplus t_8$, $t_{10} = 1 \wedge 4$, $t_{11} = t_2 \oplus t_{10}$, $t_{12} = t_9 \wedge t_{11}$ $t_{13} = t_6 \oplus t_{12}$, $t_{14} = t_5 \wedge t_{13}$, $f = t_{14} \oplus t_{10}$	4

References

1. 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)
2. Bogdanov, A., Knezevic, M., Leander, G., Toz, D., Varici, K., Verbauwhede, I.: SPONGENT: the design space of lightweight cryptographic hashing. IEEE Trans. Comput. **62**(10), 2041–2053 (2013)
3. Peralta, R.: Circuit minimization work, January 2014. http://cs-www.cs.yale.edu/homes/peralta/circuitstuff/cmt.html
4. Feldhofer, M., Wolkerstorfer, J., Rijmen, V.: AES implementation on a grain of sand. IEE Proc. Inf. Secur. **152**(1), 13–20 (2005)
5. Hamalainen, P., Alho, T., Hannikainen, M., Hamalainen, T.D.: Design and implementation of low-area and low-power AES encryption hardware core. In: Proceedings of the 9th EUROMICRO Conference on Digital System Design, DSD '06, pp. 577–583. IEEE Computer Society, Washington, DC (2006)
6. 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)
7. Boyar, J., Peralta, R.: A small depth-16 circuit for the AES S-box. In: Gritzalis, D., Furnell, S., Theoharidou, M. (eds.) SEC 2012. IFIP AICT, vol. 376, pp. 287–298. Springer, Heidelberg (2012)
8. Saarinen, M.-J.O.: Chosen-IV statistical attacks on estream ciphers. In: Malek, M., Fernández-Medina, E., Hernando, J. (eds.) SECRYPT, pp. 260–266. INSTICC Press (2006)
9. Boyar, J., Peralta, R.: A new combinational logic minimization technique with applications to cryptology. In: Festa, P. (ed.) SEA 2010. LNCS, vol. 6049, pp. 178–189. Springer, Heidelberg (2010)

10. Courtois, N., Hulme, D., Mourouzis, T.: Solving circuit optimisation problems in cryptography and cryptanalysis (2011)
11. Courtois, N., Hulme, D., Mourouzis, T.: Multiplicative complexity and solving generalized brent equations with SAT solvers. In: COMPUTATION TOOLS 2012, The Third International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking, pp. 22–27 (2012)
12. Boyar, J., Find, M., Peralta, R.: Four measures of nonlinearity. In: Spirakis, P.G., Serna, M. (eds.) CIAC 2013. LNCS, vol. 7878, pp. 61–72. Springer, Heidelberg (2013)
13. Boyar, J., Peralta, R., Pochuev, D.: On the multiplicative complexity of Boolean functions over the basis $(\wedge, \oplus, 1)$. Theor. Comput. Sci. **235**(1), 43–57 (2000)
14. Berlekamp, E.R., Welch, L.R.: Weight distributions of the cosets of the (32, 6) Reed-Muller code. IEEE Trans. Inf. Theory **18**(1), 203–207 (1972)
15. Fuller, J.E.: Analysis of affine equivalent boolean functions for cryptography. Ph.D. thesis, Queensland University of Technology (2003)
16. Maiorana, J.A.: A classification of the cosets of the Reed-Muller code R(1,6). Math. Comput. **57**(195), 403–414 (1991)
17. Braeken, A., Borissov, Y., Nikova, S., Preneel, B.: Classification of Boolean functions of 6 variables or less with respect to some cryptographic properties. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 324–334. Springer, Heidelberg (2005)
18. Hou, X.-D.: AGL (m, 2) acting on R (r, m)/R (s, m). J. Algebra **171**(3), 927–938 (1995)
19. Carlet, C.: Boolean functions for cryptography and error correcting codes. In: Crama, Y., Hammer, P.L. (eds.) Boolean Models and Methods in Mathematics, Computer Science and Engineering, chapter 8. Cambridge University Press, Cambridge (2010)
20. Uyan, E.: Analysis of Boolean Functions with respect to Walsh Spectrum. Ph.D. thesis, Middle East Technical University (2013)
21. Schnorr, C.-P.: The multiplicative complexity of Boolean functions. In: AAECC, pp. 45–58 (1988)
22. Mirwald, R., Schnorr, C.-P.: The multiplicative complexity of quadratic Boolean forms. Theor. Comput. Sci. **102**(2), 307–328 (1992)
23. Boyar, J., Matthews, P., Peralta, R.: Logic minimization techniques with applications to cryptology. J. Cryptology **26**(2), 280–312 (2013)

A Flexible and Compact Hardware Architecture for the SIMON Block Cipher

Ege Gulcan(✉), Aydin Aysu, and Patrick Schaumont

Secure Embedded Systems,
Center for Embedded Systems for Critical Applications,
Bradley Department of ECE,
Virginia Tech, Blacksburg, VA 24061, USA
{egulcan,aydinay,schaum}@vt.edu

Abstract. SIMON is a recent, light-weight block cipher developed by NSA. Previous work on SIMON shows that it is a very promising alternative of AES for resource-constrained platforms. While SIMON offers a range of block sizes and key lengths, a straightforward implementation would select fixed values in order to achieve a compact design. In contrast, we propose a flexible hardware architecture on FPGAs that still preserves the compactness of SIMON. The proposed implementation can execute all configurations of SIMON, and thus provides a versatile architecture that enables adaptive security using a variable key-size. Moreover, it also reduces the inefficiency of encrypting slightly longer messages by supporting a variable block-size. The implementation results show that the proposed architecture occupies 90 and 32 slices on Spartan-3 and Spartan-6 FPGAs, respectively. To our best knowledge, these area results are smaller than other block ciphers of similar security level. Furthermore, we also quantify the cost of flexibility and show the trade-off between the security level, throughput and area.

Keywords: Lightweight cryptography · Block ciphers · Flexible architectures · SIMON · FPGA

1 Introduction

Block ciphers are the building blocks of secure systems as they enable sending a message over a non-secure medium. These ciphers perform symmetric-key encryption by mapping a block of input plaintext to an output ciphertext using a secret key. Once the ciphertext is generated, it can only be decrypted back into the plaintext by using exactly the same secret key. Rijndael is the most widely used block cipher algorithm and it is used as the Advanced Encryption Standard (AES) [8].

Even though Rijndael serves as the AES, its area-cost restricts its use in resource-critical domains like RFID tags. This is where lightweight cryptography shines. The goal of lightweight cryptography is to minimize the area of implementing and executing an operation while preserving similar or slightly reduced

© Springer International Publishing Switzerland 2015
T. Eisenbarth and E. Öztürk (Eds.): LightSec 2014, LNCS 8898, pp. 34–50, 2015.
DOI: 10.1007/978-3-319-16363-5_3

levels of security. With the aim of reducing the area of the AES, two alternatives named PRESENT and CLEFIA were previously developed and later standardized by ISO [10]. Likewise, DARPA has an ongoing SHIELD project that is targeted towards tackling counterfeit electronics [6]. The goal of the project is to enable supply-chain management by means of a light-weight secure hardware of 100 micron $\times$ 100 micron size [7]. Therefore, there are important incentives to build the basic encryption block that are much smaller than the available ones. SIMON is such an alternative which is optimized for compact hardware implementations [2]. Aysu *et al.* showed that SIMON can break the area records of block ciphers on FPGAs [1]. They implement a fixed 128/128 configuration of SIMON that can only encrypt blocks of 128-bit messages using a 128-bit key. However, the design space of digital systems are not solely composed of fixed elements, and flexibility among others is an important design dimension.

1.1 Motivation

Security is a new design dimension for digital systems [12]. Schaumont *et al.* labels this dimension as Risk and shows that flexibility, performance and risk are the main design dimensions of secure embedded systems [18]. Furthermore, they argue that a good design should consider the trade-offs between these dimensions. In that framework, performance refers to the capability of the system for a given target metric (throughput, energy-efficiency, area, etc.), risk is the potential for loss, and flexibility is the ability to (re)define the system parameters and behavior. The dimension of flexibility is even more important especially for applications with a diverse set of requirements. Wireless sensor networks (WSN) are an outstanding example for this scenario. WSN typically consist of a large number of devices (nodes) that are one-time programmed and deployed in the field. The nodes run for long periods of time without human intervention.

A common practice of flexibility is to implement adaptive security for WSN. Younis *et al.* proposes an adaptive security provision for wireless sensor nodes [23]. They propose an efficient protocol in which the encryption strength (key-size) varies between 32-bits to 128-bits depending on the trust level of the nodes. Obviously, if a node is more trusted, an encryption with a lower level of security allows computation savings. Wang *et al.* argues a similar case for computation savings where the sensitive data within the network is encrypted with a higher security level, while the less important information is encrypted using shorter keys [21]. Sharma *et al.* claims that the application diversity of WSN ranges from military surveillance to agriculture farming, each of which requiring a different set of minimal security mechanisms [19]. Then, they present a comprehensive security framework that can provide security services for a variety of applications. Finally, Portilla *et al.* provides a case study on FPGAs using the Elliptic Curve Cryptography and proposes a solution for a public-key based adaptable security on WSN [17].

Cook *et al.* approaches flexibility from another perspective [5]. If an input plaintext is even one-bit larger than the encryption block-size n, it has to be padded to $2n$ and the encryption should run more than once. Therefore, they

introduce an elastic block cipher that improves the inefficiency by allowing a variable block-size. This methodology uses a fixed key-size with a variable block-size.

Our solution combines the merits of both visions. We propose an architecture that can have both variable block-size and key-size. Using such a flexible architecture enables a single device to offer adaptable security for a variety of applications, or multiple levels of security within an application. It can also reduce the redundancy of slightly longer messages by changing the encryption block-size. Our unified architecture also minimizes the licensing/certification efforts since we use a single design for many different use-cases. The complex cryptographic module validation programs like NIST CMVP [16] also make the single hardware running all configurations advantageous over the collection of many that can execute a single configuration. Yet, the proposed architecture is still very compact which makes it very suitable for light-weight applications.

From a design methodology perspective, the proposed hardware provides flexibility (at the expense of area and throughput) to the system by enabling on-the-fly security configuration management. It also allows a trade-off between the performance and risk. Our results show that the system can increase the security from 64-bits to 256-bits (from toy-settings to high-profile security) with a throughput degradation of a factor of 2. Moreover, to our best knowledge, the proposed flexible hardware architecture of SIMON is still smaller than other block ciphers of similar security level.

1.2 Organization

The rest of the paper is organized as follows. Section 2 gives a brief overview of SIMON block cipher and its configurations. Section 3 highlights the methodology behind the compact block cipher architectures and how to extend it for flexibility. Section 4 shows the implementation results and presents the trade-off between flexibility, performance and risk. Section 5 concludes the paper and comments on possible future extensions.

2 SIMON Block Cipher

SIMON is a Feistel-based lightweight block cipher recently published by NSA, targeted towards compact hardware implementations [2]. SIMON has ten configurations optimized for different block and key sizes providing a flexible level of security. Table 1 shows the parameters for all configurations of SIMON. The word size n is the bit length of each word in the Feistel network, which makes the block size to be $2n$. The key length is defined as a multiple of the Feistel word size, and the parameter m indicates the number of Feistel words in a key. Security configuration is a new parameter that we introduce to select the desired configuration of SIMON.

Table 1. Simon parameters

Security configuration	Block size (2n)	Key size	Word size (n)	Key words (m)	Rounds
1	32	64	16	4	32
2	48	72	24	3	36
3	48	96	24	4	36
4	64	96	32	3	42
5	64	128	32	4	44
6	96	96	48	2	52
7	96	144	48	3	54
8	128	128	64	2	68
9	128	192	64	3	69
10	128	256	64	4	72

2.1 Round Function

Figure 1 shows the round function for all configurations of SIMON. X_{upper} and X_{lower} respectively denote the upper and lower words of the block and they are n-bits each. These two words hold the initial input plaintext and the output after each round is executed. The round function consists of bitwise AND, bitwise XOR, and circular shift left operations. In each round, shifting and bitwise AND operations are performed on the upper word and it is XORed with the lower word and the round key. The resulting value is written back to the upper word while its content is transferred over to the lower word. The round function continues to run repeatedly until the desired number of rounds is reached.

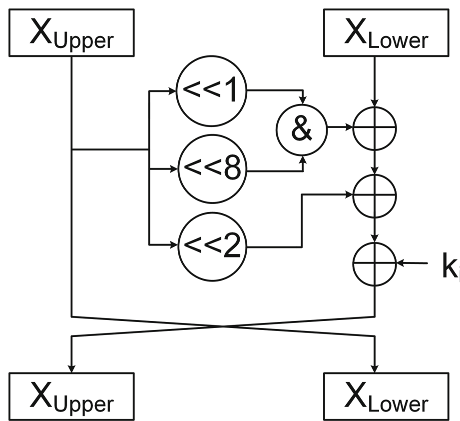

Fig. 1. SIMON round function

2.2 Key Expansion

SIMON block cipher needs unique keys for each round and the key expansion function generates these round keys. Unlike the round function, there are three different configurations of key expansion as the number of words in a key can be 2, 3 and 4 depending on the configuration. Figure 2 shows the key expansion functions for three different key lengths, corresponding to two, three or four Feistel words respectively ($m = 2$, 3 or 4). The block K_i holds the round key for the ith round. For $m = 2$ and $m = 3$, the logical operations of the key expansion function are identical. The most significant word is circular shifted right by 3 and 4, and it is XORed with the least significant word and the round constant z_i. For $m = 4$, there is an extra step where the most significant word (K_{i+3}) is circular shifted right by 3, XORed with K_{i+1}, then circular shifted right by 1 and XORed with the least significant word and the round constant. At the end of each key expansion, the new round key is written into the most significant word, and all the words are shifted one word right. As K_i is the key used in the current round, it will no longer be needed and is overwritten. The key expansion function has a sequence of one bit round constants used for eliminating slide properties and circular shift symmetries. There are five different round constant sequences uniquely tuned for each configuration to provide a cryptographic separation between the different configurations.

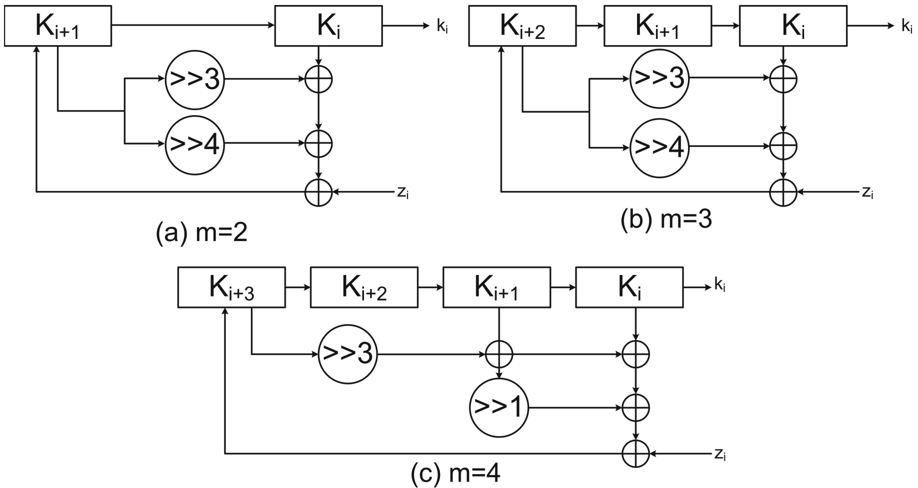

Fig. 2. SIMON key expansions

3 Hardware Implementation

When implementing a block cipher on hardware, there are several parallelism choices (bit level, round level, and encryption level) that affect the area and

throughput of the design. In bit level parallelism, the input size of the operators range from one bit to n-bits where n is the block size. In round level parallelism, we can have one round up to r-rounds per clock cycle where r is the total number of rounds of the block cipher. Finally, in encryption level parallelism, we can have one encryption engine up to e encryption engines where e is the maximum number of engines that can fit in our area constraints. Depending of the chosen levels of parallelism, our design space will range from p parallel encryptions per clock cycles to one bit of one round of one encryption engine per clock cycle. In order to keep the area of our design as low as possible, we used the lowest parallelism level of one bit of one round of one engine, which is also called the bit-serial implementation.

3.1 Bit-Serial

Figure 3 shows the details of the round (a) and key expansion functions (b,c,d) of the bit serial SIMON. The current state holds the words that are used in the current round and the next state holds the words that are generated after the execution of the first round and will be used in the next round. Both of these states share the same set of memory elements and they are overwritten in every round. In the key expansion functions, K_i denotes the key that will be used in the ith round. The highlighted bits indicate the bits that are processed at the first clock cycle of each round.

Both the key expansion and the round function consist of two phases: Compute and Transfer. The compute phase reads the necessary bits from the current state, performs logic operations on them and writes the resulting bit into the upper block of the next state, while the transfer phase copies the contents of a word in the current states to a lower word in the next state. For the key expansion, there are three different functions depending on the number of key words. The compute phase is the same for $m = 2$ and $m = 3$ where only three bits are necessary from upper and lower words. For $m = 4$, two additional bits are required from the word K_{i+1} to compute the next state bit. The number of transfer phases required to finish one expansion also changes with the key words number.

The bit serial implementation of the SIMON block cipher fits very well into the resources of an FPGA as we can use the Look Up Tables (LUT) as memory elements. In a Spartan-3 family FPGA, each LUT can be configured as an 16×1 Shift Register LUT (SRL), in which we can store the words of the round and key expansion functions. Since we are reading from and writing into the SRL one bit per clock cycle, we will call them FIFOs throughout this paper. By using these FIFOs we can overlap the compute and transfer phases to process one bit in every clock cycle.

3.2 Round Function

The round function of the SIMON block cipher is the same for all ten configurations except for the size of the memory elements (words). In the Feistel network

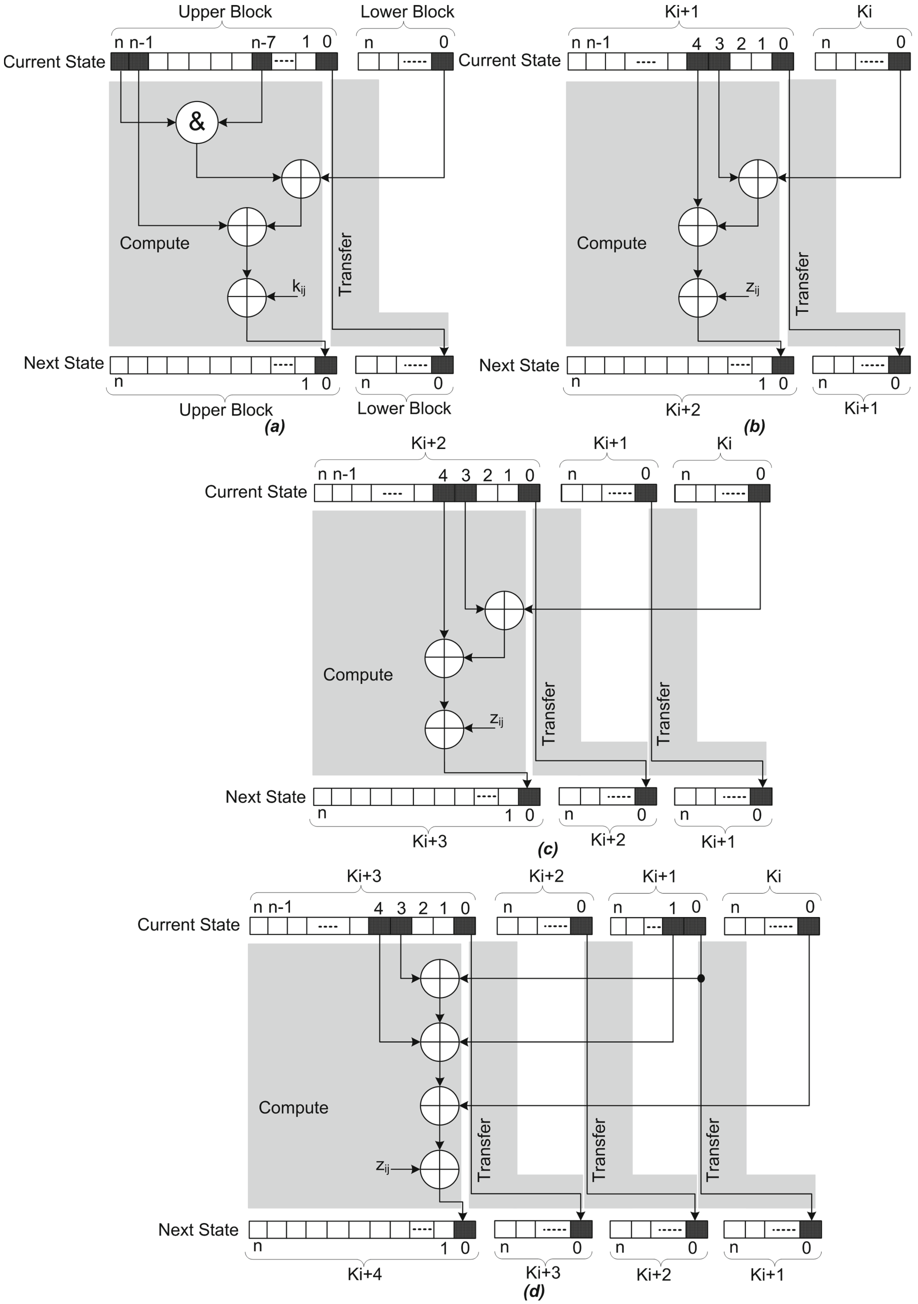

Fig. 3. (a) SIMON Bit-serial round function, (b) SIMON Bit-serial key expansion for $m = 2$, (c) SIMON Bit-serial key expansion for $m = 3$, (d) SIMON Bit-serial key expansion for $m = 4$

of the round function, the block is separated into two words, each keeping one half of the complete block. As the block size changes with different versions, the size of the FIFOs holding these words also change accordingly. In order to have a round function that can work with any of the ten versions, we need to have a flexible length of FIFOs.

Figure 4 shows the bit serial implementation of the flexible round function of SIMON. There are two groups of FIFOs named FIFO_1 and FIFO_2, which hold the upper and lower words of the block. Each group is divided into subsections of FIFOs with different sizes, connected together through multiplexers. The sizes of the subsection FIFOs are selected such that each additional FIFO increases the total size to be equal to the desired word size. FIFO_1 is smaller than FIFO_2 as the eight most significant bits of the upper word are stored in the Shift Registers Up or Down. These shift registers are required due to the circular shift pattern of the round function. As we are using one bit input-output FIFOs, we cannot access the intermediate bits. Therefore, the registers store the first eight bits in flip-flops to enable parallel access. According to the security configuration input, multiplexers select the required size of the FIFOs for both the upper and lower words and route the incoming data to the correct subsection of FIFOs.

Each FIFO has a two input multiplexer at its input that bypasses the unused FIFOs and routes the FIFO group input to the desired subsection FIFO. When input '0' is selected, the FIFO group input is connected to the subsection FIFO and when input '1' is selected, the next FIFOs output is connected. Figure 5 shows the required FIFO numbers for all security configurations.

For example, if the security configuration input is 1, the round function use FIFO_1_0 and FIFO_2_0 while the rest of the FIFOs are grounded. The output of FIFO_1_0 is connected to the input of FIFO_2_0 to perform the transfer operation, and the data coming from SRU or SRD (depending on the round number) is connected to the input of FIFO_1_0. When the security configuration input changes to 2, the word size increases from 16 bits to 24 bits. Therefore, one additional FIFO of size 8 is needed to store the upper and lower words. The multiplexers at the inputs of FIFO_1_0 and FIFO_2_0 now select the output of the FIFOs to their left (select input 1), and the FIFO group inputs are routed to FIFO_1_1 and FIFO_2_1 (select input 0).

One important aspect of the bit serial implementation is the use of two sets of shift registers named Shift Register Up (SRU) and Shift Register Down (SRD). As the round function of SIMON requires three circular shift left operations (1, 2 and 8) on the upper block, the current state bits required to compute the next state bit do not go in a sequentially ordered manner. For example, when the block size n is 32, in order to compute the bit #0 of the next state, we need to use the bits #31, #30 and #24 of the upper block of the current state. However, the new computed bit #0 should also be stored in the same memory element of the upper block which causes a conflict. We need to use the bit #0 of the current state to compute the bit #1 of the next state so we cannot overwrite it yet. In order to solve this problem, we implemented the ping pong registers SRU and SRD. In the even numbered rounds, the output of the LUT is written

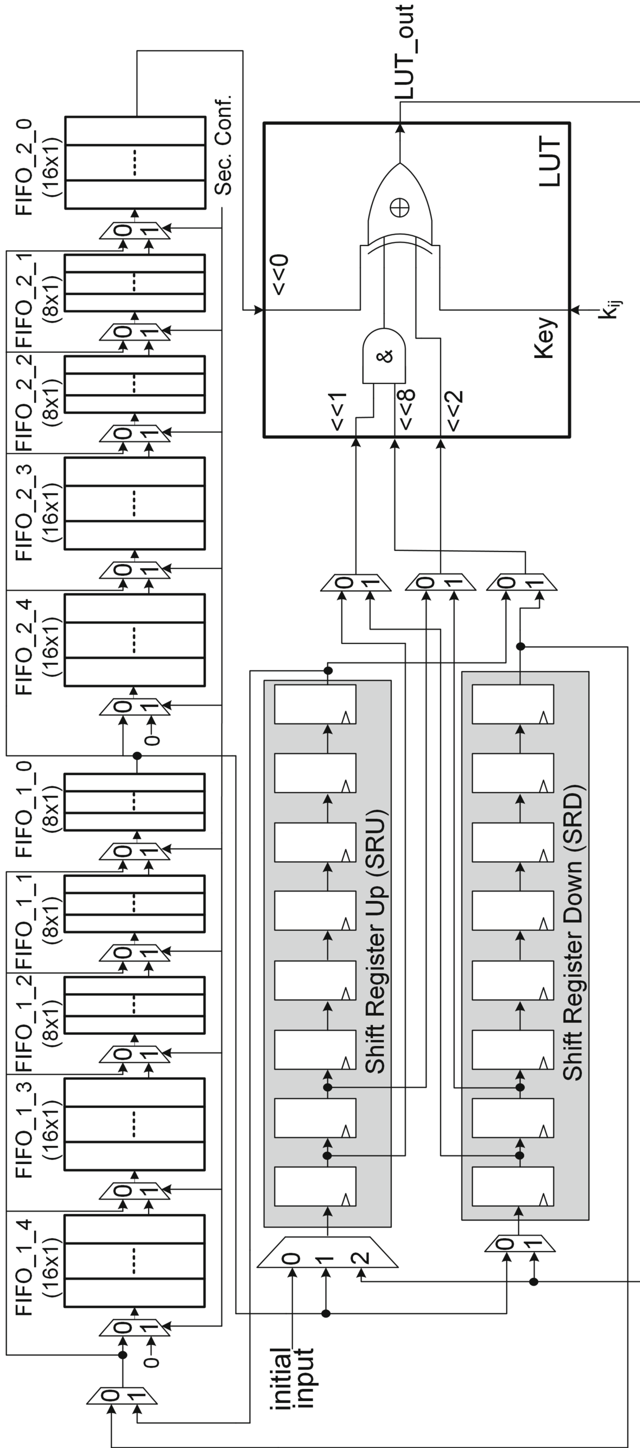

Fig. 4. SIMON Bit-serial flexible round function

to the SRD and the output of the FIFO_1 is written to the SRU. Also for the first eight bits, the input of the FIFO_1 is connected to the output of SRU and for the rest it is connected to the output of SRD. In the odd numbered rounds, we interchange the usage of SRU and SRD. By using this technique, we append the least significant eight bits of the upper block to its most significant bits to solve the circular shift problem and we can finish one round in n clock cycles.

3.3 Key Expansion

Unlike the round function, there are three different key expansion functions depending on the block size and the key size. Figure 6 shows the flexible bit-serial key expansion of SIMON. There are four groups of FIFOs that store the round keys and similar to the round function, they are divided into subsection FIFOs in order to achieve a flexible size. Since the logical operations for the key word number $m = 4$ are different, we need two LUTs to perform the different key expansion function operations. For $m = 2$ and $m = 3$ the hardware uses LUT2 for the logical operations, while for $m = 4$ it uses LUT1. A LUT based ROM stores the round constants and according to the security configuration input, the multiplexer selects the appropriate sequence.

Another difference of key generation is the dependence of the FIFO group activity to the security configuration input. As there are three possible numbers of key words ($m = 2, 3, 4$), not only the number of subsection FIFOs but also the number of FIFO groups utilized should be flexible. The number of FIFO groups required for each security configuration is equal to the number of key words m of the selected configuration. For $m = 2$ the hardware only uses FIFO_0 and FIFO_3. When $m = 3$ it also utilizes FIFO_2, and if $m = 4$ it enables all four FIFO groups. Additionally, the number of subsection FIFOs changes with the key size. Figure 5 gives the details of which FIFOs are used for all the security configurations.

As it can be seen in Fig. 6, FIFO_3 and FIFO_1 have four (FIFO_3_FF) and two (FIFO_1_FF) additional flip-flops at their outputs, respectively. The necessity of these separate flip-flops come from the circular shift operations of the key expansion function. We used the same technique to overcome the circular

Security Conf.	Datapath FIFO No.										Key Expansion FIFO No.																		
	1_0	1_1	1_2	1_3	1_4	2_0	2_1	2_2	2_3	2_4	0_0	0_1	0_2	0_3	0_4	1_0	1_1	1_2	1_3	2_0	2_1	2_2	2_3	2_4	3_0	3_1	3_2	3_3	3_4
1	x					x					x					x				x					x				
2	x	x				x	x				x	x								x	x				x	x			
3	x	x				x	x				x	x				x	x			x	x				x	x			
4	x	x	x			x	x	x			x	x	x							x	x	x			x	x	x		
5	x	x	x			x	x	x			x	x	x			x	x	x		x	x	x			x	x	x		
6	x	x	x	x		x	x	x	x		x	x	x	x											x	x	x	x	
7	x	x	x	x		x	x	x	x		x	x	x	x						x	x	x	x		x	x	x	x	
8	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x										x	x	x	x	x
9	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x					x	x	x	x	x	x	x	x	x	x
10	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x

Fig. 5. FIFO usage schedule

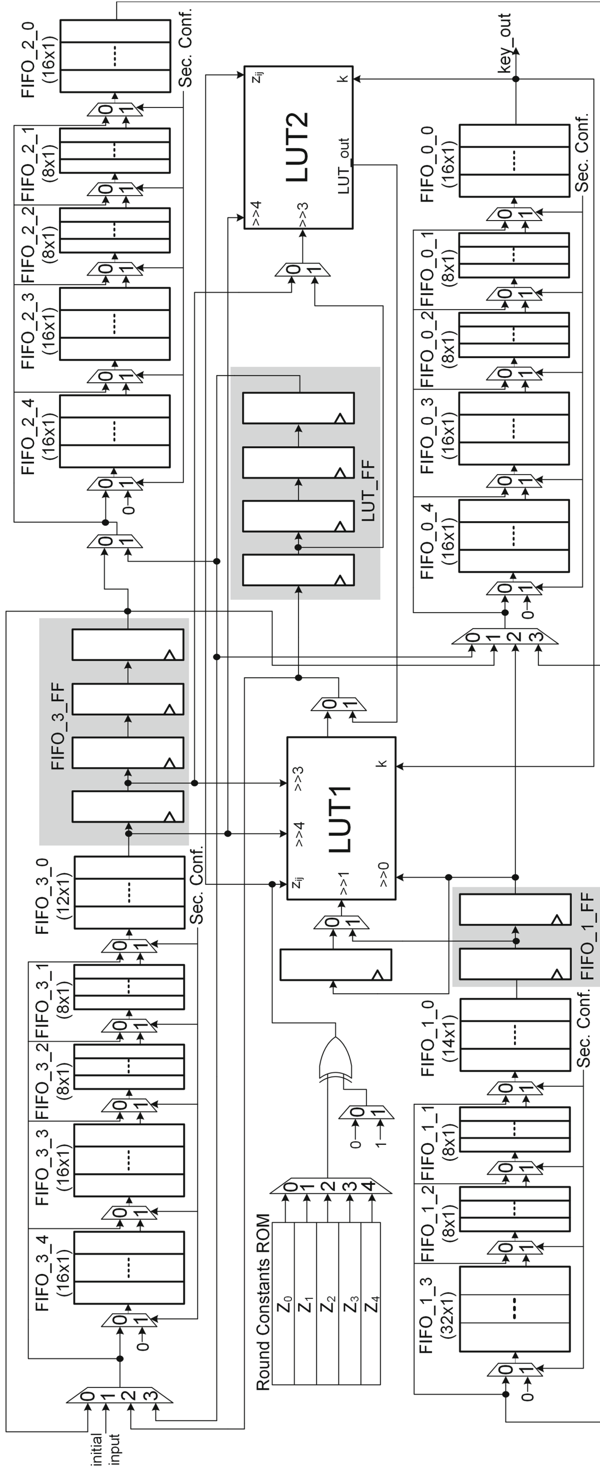

Fig. 6. SIMON Bit-serial flexible key expansion

shift patterns of the round function, but this time we put the flip-flops at the end of the FIFOs, as the key expansion function uses circular shift right, rather than left. At the first four clock cycles of each round, the input for FIFO_3 is the output of FIFO_3_FF. This way, the least significant four bits of the word are appended into the most significant four bits. At the same time, the output of LUT has to be connected to the same memory element which causes a conflict. Therefore, the architecture uses another set of four flip-flops (LUT_FF) that store the output of the LUT for the first four clock cycles. After this period ends, FIFO_3 can directly store the output of LUT since appending more bits is not necessary. At the beginning of the second round, the content of FIFO_3_FF is not fresh as it contains the four bits from the previous round. Therefore, FIFO_3_FF will only be active in the first round. LUT_FF takes its responsibility to append the first four bits to FIFO_3 and also store the outputs of the LUT. Note that in this discussion we do not mention the security configuration input because no matter what the configuration is, FIFO_3 will use this scheduling, only the size of the subsection FIFOs will change.

For $m = 2$, FIFO_3 group stores the upper key word and FIFO_0 group stores the lower one. The transfer operation performs data transfer operation between these two FIFOs. Since the LUT_FF stores the first four bits of each new computed key, during this period the input of FIFO_0 is LUT_FF, and for the rest of the clock cycles, it is FIFO_3. For $m = 3$, in addition to FIFO_0 and FIFO_3, the hardware also utilize FIFO_2 group to store the additional key word. There are two concurrent transfer operations; the first from FIFO_3 to FIFO_2, and the second one from FIFO_2 to FIFO_0. LUT2 computes the logical operations for both $m = 2$ and $m = 3$.

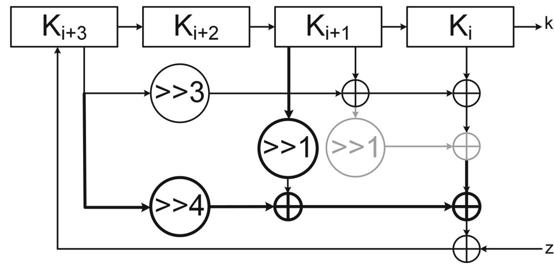

Fig. 7. SIMON modified key expansion function for $m = 4$

Executing a circular shift operation after a logical operation is problematic for bit-serialized implementation. Therefore, the original form of the key expansion for $m = 4$ is not suitable for a bit-serial implementation because it requires a circular shift right operation after the XOR of K_{i+3} and K_{i+1}. In order to solve this problem, we modify the key expansion function for $m = 4$. Figure 7 shows the required transformation. The gray regions highlight the original operations which are replaced with the bold regions. Originally, the output of the XOR

operation has two fanouts, one going directly to another XOR with K_i and the second one to a circular shift operation. We moved the circular shift right by 1 operation from the output to the inputs of the XOR. The XOR from K_{i+3} was originally circular shifted right by 3 and when we shift it one more after the modification, it becomes a circular shift right by 4. Similar to the functionality of FIFO_3_FF, FIFO_1_FF enables the circular access pattern of $m = 4$.

4 Implementation Results

The proposed hardware architecture is written in Verilog HDL. The Verilog HDL RTL codes are synthesized to the Xilinx Spartan-3 s50 FPGA using a speed grade of −5, and to the Spartan-6 lx4 FPGA using a speed grade −3. Then, the resulting netlists are placed and routed to the same FPGAs using Planahead. In order to minimize the slice count, we hand-pick our design elements and assign their mapping into the slices.

4.1 Area

Comparison with Other Block Ciphers. Figure 8 shows hardware resource utilization of our architecture and the previous work. We have compared our work with the smallest version of AES [4], as well as alternative compact block cipher implementations such as PRESENT [22], HIGHT [22], SEA [14], XTEA [11], CLEFIA [3] and ICEBERG [20]. In order to have a fair comparison, we map our hardware into the same FPGA (Spartan-3) with the previous work, but we also show the occupied area on a more recent FPGA like Spartan-6. The proposed hardware occupies 90 and 32 slices on a Spartan-3 and a Spartan-6 FPGA, respectively. Out of all these implementations, our hardware architecture is the only one that provides the flexibility, whereas the rest of them use a fixed key and block size. Yet, our flexible hardware architecture is still smaller than all block ciphers. These results show that our bit-serial design methodology and our back-end tool-optimization was able to achieve very compact hardware instances while still enabling the flexibility.

Comparison with Other Flexible Architectures. There are several architectures in literature that implement the multiple configurations of AES. However, none of them where targeted for light-weight platforms. AES has three configurations, AES-128, AES-192, and AES-256 all use 128-bit block size with 128, 192 and 256 bit key-size, respectively. McLoone *et al.* proposes an architecture that can perform all configurations using 4681 slices [15]. Li *et al.* later optimizes this implementation and reduces the slice count to 3223 slices [13].

Comparison with Commercial Soft-Core Processors. One alternative way to implement a flexible encryption engine on an FPGA is through a soft-core processor. Xilinx provides Microblaze whereas Altera proposes NIOS as a commercial soft-core processor. These processors execute software that enables

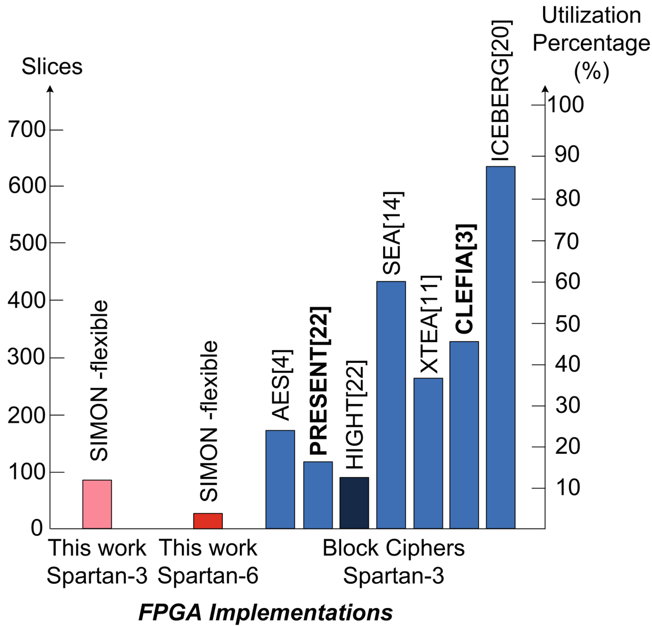

Fig. 8. Occupied slices and the resource utilization ratio of flexible SIMON vs. previous work.

the capability of running all configurations. However, the minimum area-cost of a NIOS and Microblaze is approximately 700 logic elements (logic element is 1 LUT + 1 register) and 600 slices, respectively. Picoblaze is an area-optimized Xilinx processor that can bring the area-cost down to 96 slices and 1 BRAM, which is still higher than our memory-free architecture.

4.2 Performance vs. Risk Trade-off

Figure 9 shows the trade-off between the performance and the risk. As we increase the size of the key, we decrease the risk of the system. However, we also increase the total time of computation because SIMON requires more rounds to complete, and the bit-serial architecture requires more clock cycles to finish one round. For example, if the system selects the security configuration 1, it takes 32 rounds to complete the encryption of a 32-bit block and one round is processed in 16 clock cycles. Therefore, the throughput of the encryption is 5.27 Mbps. On the other hand, the system will be using a key-length of 64-bits which can be regarded as a toy-setting since dedicated machines like COPACOBANA can break a block cipher with 57-bits in less than a week [9]. If the system changes its settings to the security configuration of 10, the key size will be 256-bits. Hence, the risk will be much lower, but the throughput will decrease by a factor of 2.

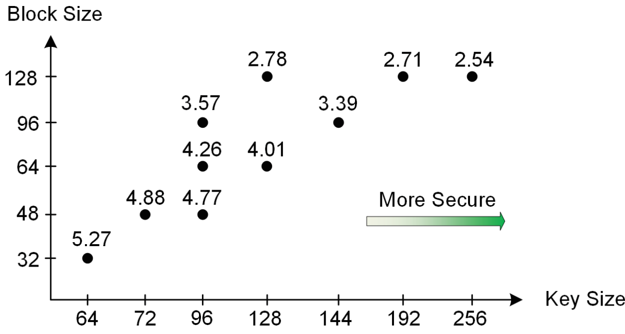

Fig. 9. Throughput (Mbps) vs. the security configuration of SIMON

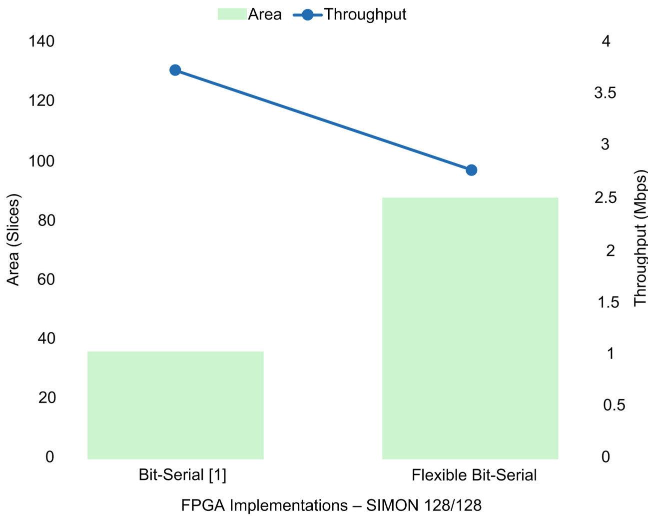

Fig. 10. The cost of flexibility on area and throughput

4.3 Flexibility vs. Performance Trade-off

Flexibility comes at the expense of performance. Figure 10 illustrates the cost of implementing the flexible architecture. We compare our flexible architecture running at the security configuration of 8 to the results of Aysu *et al.*, as they both use 128-bit key size and block size [1]. Since the proposed flexible architecture has to support all available configurations, including the ones that has larger keys and block sizes, the slice count is approximately three times of the fixed implementation. Even though the required clock cycles to complete the encryption is equal for the two architectures, a larger circuit causes longer interconnect delays and a lower maximum achievable frequency. Therefore, compared to the fixed implementation, the throughput of the flexible architecture degrades by 23 %.

5 Conclusion and Future Work

In this paper, we propose a flexible and compact architecture for the block cipher SIMON. SIMON is a very promising alternative of AES for resource-constrained platforms and we show that the bit-serialized flexible implementation of SIMON is still smaller than other block ciphers. The proposed architecture can implement all configurations of SIMON and enables on-the-fly security configuration management. Thus, we propose a light-weight, yet flexible and adaptive solution for secure systems. We also show the trade-offs that a designer can utilize regarding the flexibility, performance and risk. A further extension of this work may be proposing a complete system that can use the proposed architecture in an adaptive security protocol. Such a protocol provides different levels of security to its users based on some pre-defined criteria or may scale-up/down the risk on-the-fly, to meet the real-time performance requirements.

Acknowledgments. This project was supported in part by the National Science Foundation grant no 1115839.

References

1. Aysu, A., Gulcan, E., Schaumont, P.: SIMON says: Break area records of block ciphers on FPGAs. IEEE Embed. Syst. Lett. **6**(2), 37–40 (2014)
2. Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., Wingers, L.: The SIMON and SPECK families of lightweight block ciphers (2013)
3. Chaves, R.: Compact CLEFIA implementation on FPGAs. In: Athanas, P., Pnevmatikatos, D., Sklavos, N. (eds.) Embedded Systems Design with FPGAs, pp. 225–243. Springer, New York (2013). http://dx.doi.org/10.1007/978-1-4614-1362-2_10
4. Chu, J., Benaissa, M.: Low area memory-free FPGA implementation of the AES algorithm. In: 2012 22nd International Conference on Field Programmable Logic and Applications (FPL), pp. 623–626, August 2012
5. Cook, D.L.: Elastic block ciphers. Ph.D. thesis, Columbia University (2006)
6. DARPA: SHIELD: supply chain hardware integrity for electronics defense proposers day, February 2014
7. DARPA: Tiny, cheap, foolproof: Seeking new component to counter counterfeit electronics, February 2014. http://www.darpa.mil/NewsEvents/Releases/2014/02/24.aspx
8. FIPS PUB 197: AES: Advanced encryption standard. Federal Information Processing Standards Publication (2001)
9. Guneysu, T., Kasper, T., Novotny, M., Paar, C., Rupp, A.: Cryptanalysis with COPACOBANA. IEEE Trans. Comput. **57**(11), 1498–1513 (2008)
10. ISO/IEC 29192–2:2012: Information technology - security techniques - lightweight cryptography - part 2: Block ciphers (2012)
11. Kaps, J.-P.: Chai-Tea, Cryptographic Hardware Implementations of xTEA. In: Chowdhury, D.R., Rijmen, V., Das, A. (eds.) INDOCRYPT 2008. LNCS, vol. 5365, pp. 363–375. Springer, Heidelberg (2008)
12. Kocher, P., Lee, R., McGraw, G., Raghunathan, A.: Security as a new dimension in embedded system design. In: Proceedings of the 41st Annual Design Automation Conference, DAC 2004, pp. 753–760. ACM, New York (2004). http://doi.acm.org/10.1145/996566.996771, moderator-Ravi, Srivaths

13. Li, H.: Efficient and flexible architecture for AES. IEE Proc. Circuits, Devices Syst. **153**(6), 533–538 (2006)
14. Mace, F., Standaert, F.X., Quisquater, J.J.: FPGA implementation(s) of a scalable encryption algorithm. IEEE Trans. Very Large Scale Integration (VLSI) Systems **16**(2), 212–216 (2008)
15. McLoone, M., McCanny, J.: Generic architecture and semiconductor intellectual property cores for advanced encryption standard cryptography. IEE Proc. Comput. Digital Tech. **150**(4), 239–244 (2003)
16. NIST: Cryptographic Module Validation Program Management Manual, May 2014. http://csrc.nist.gov/groups/STM/cmvp/documents/CMVPMM.pdf
17. Portilla, J., Otero, A., de la Torre, E., Riesgo, T., Stecklina, O., Peter, S., Langendrfer, P.: Adaptable security in wireless sensor networks by using reconfigurable ECC hardware coprocessors. In: IJDSN 2010 (2010). http://dblp.uni-trier.de/db/journals/ijdsn/ijdsn2010.html#PortillaOTRSPL10
18. Schaumont, P., Aysu, A.: Three design dimensions of secure embedded systems. In: Gierlichs, B., Guilley, S., Mukhopadhyay, D. (eds.) SPACE 2013. LNCS, vol. 8204, pp. 1–20. Springer, Heidelberg (2013). http://dx.doi.org/10.1007/978-3-642-41224-0_1
19. Sharma, K., Ghose, M.: Cross layer security framework for wireless sensor networks. Int. J. Secur. Appl. **5**(1), 35–52 (2011)
20. Standaert, F.X., Piret, G., Rouvroy, G., Quisquater, J.J.: FPGA implementations of the ICEBERG block cipher. In: International Conference on Information Technology: Coding and Computing, ITCC 2005, vol. 1, pp. 556–561 (2005)
21. Wang, Y., Attebury, G., Ramamurthy, B.: A survey of security issues in wireless sensor networks. IEEE Commun. Surv. Tutorials **8**(2), 2–23 (2006)
22. Yalla, P., Kaps, J.: Lightweight cryptography for FPGAs. In: International Conference on Reconfigurable Computing and FPGAs, ReConFig 2009, pp. 225–230 (2009)
23. Younis, M., Krajewski, N., Farrag, O.: Adaptive security provision for increased energy efficiency in wireless sensor networks. In: IEEE 34th Conference on Local Computer Networks, LCN 2009, pp. 999–1005, October 2009

AES Smaller Than S-Box

Minimalism in Software Design on Low End Microcontrollers

Mitsuru Matsui and Yumiko Murakami(✉)

Information Technology R&D Center, Mitsubishi Electric Corporation, Chiyoda-ku, Japan
Matsui.Mitsuru@ab.MitsubishiElectric.co.jp,
Murakami.Yumiko@cw.MitsubishiElectric.co.jp

Abstract. This paper explores state-of-the-art software implementations of "size-minimum" AES on low-end microcontrollers. In embedded environments, reducing memory size often has priority over achieving faster speed. Some recent lightweight block ciphers can be implemented in 200 to 300 ROM bytes, while the smallest software implementation of AES including key scheduling, encryption and decryption is, as far as we know, around 1 K ROM bytes.

The first purpose of this study is to see how small AES could be. To do this, we aggressively minimize code and data size of AES by introducing a ring multiplication for computing the S-box without any lookup table, a compact algorithm for embedding MixColumns into InvMixColumns, and a tiny loop for processing AddRoundKey, ShiftRows and SubBytes at the same time. As a result, we achieve a 192-byte AES encryption-only code and a 326-byte AES encryption-decryption code on the RL78 microcontroller. We also show that an AES-GCM core can be implemented in 429 bytes on the same microcontroller. These codes include on-the-fly key scheduling to minimize RAM size and their running time is independent of secret information, i.e. timing-attack resistant.

The second purpose of this research is to see what processor hardware architecture is suitable for implementing lightweight ciphers from a minimalist point of view. A simple-looking algorithm often results in very different size and speed figures on different low-end microcontrollers in practice, even if their instruction sets consist of similar primitive operations. We show concrete code examples implemented on four low-end microcontrollers, RL78, ATtiny, Cortex-M0 and MSP430 to demonstrate that slight differences of processor hardware, such as carry flag treatment and branch timing, significantly affect size and speed of AES.

1 Introduction

Lightweight is one of the recent keywords in cryptography, with increasing market requirements of embedded security as a background. A lot of new lightweight symmetric ciphers and hash functions have been proposed, aiming at achieving low resource occupation and at the same time maintaining high level security.

© Springer International Publishing Switzerland 2015
T. Eisenbarth and E. Öztürk (Eds.): LightSec 2014, LNCS 8898, pp. 51–66, 2015.
DOI: 10.1007/978-3-319-16363-5_4

Lightweight cryptography is in many cases studied in the context of hardware lightweight such as low energy consumption and small circuit area, but software lightweight is also getting paid attention. Some recent researches concentrate on extensive software implementation of lightweight ciphers on an embedded microcontroller [1–3].

In embedded environments, reducing memory size often has priority over achieving faster speed and it has been reported that some lightweight block ciphers can be implemented on an embedded microcontroller in extremely small 200 to 300 ROM bytes [3–5]. This paper goes deep into this direction for AES. As far as we know, the smallest software AES with 128-bit key, including key scheduling, encryption and decryption, still requires 1 K ROM bytes [3]. In fact, to create an AES code within 1. 5K ROM bytes, loop rolling is necessary inside its round function, which leads to heavy performance penalty. This explains why most of known AES implementations require at least 1.5 K ROM bytes.

Our aim is to see how small AES could be, and to achieve this goal, we aggressively try to minimize code and data size of AES. Our code does not use any lookup tables for the S-box. It is instead computed with a Galois field inversion and a matrix multiplication as in its original definition. While the matrix multiplication is not a Galois field operation, we point out that a Galois field multiplication included in the Galois field inversion is "essentially the same" as the matrix multiplication since the former is a ring operation on $GF(2)[x]/(x^8+x^4+x^3+x+1)$ and the latter is that on $GF(2)[x]/(x^8+1)$. This observation leads to a new compact logical S-box code. Note that the fact that the matrix is circular is essential.

We next show that `MixColumns` can be fully embedded in `InvMixColumns` not only in hardware [6] but in software in a very simple and compact manner. In fact, `InvMixColumns` also works as `MixColumns` by just adding one conditional jump indicating encryption or decryption into `InvMixColumns`. This greatly contributes to code reduction of AES containing both encryption and decryption. In addition, it is demonstrated that `AddRoundKey`, `ShiftRows` and `SubBytes` can be merged into a tiny loop of around 20 bytes, except the S-box logic.

As a result, we achieve a 192-byte AES encryption-only code and a 326-byte AES encryption-decryption code on the RL78 microcontroller. We also show that an AES-GCM core can be implemented in 429 bytes on RL78. These algorithms are implemented on the ATtiny microcontroller as well, and it is seen that our resultant codes are a bit larger but much faster than those on RL78. All of these codes include on-the-fly key scheduling to minimize RAM size, and their running time is independent of secret information such as key and text.

The second purpose of this research is to see what processor hardware is suitable for implementing lightweight ciphers from a minimalist point of view. It is rather common that a code based on the same algorithm exhibits very different size and speed figures on different low-end microcontrollers, even if their instruction sets consist of similar primitive operations. Many types of low-end microcontrollers have been used in real-world embedded applications, but their comparative research from a cryptographic point of view seems missing.

We show concrete examples extracted from our AES codes and implemented on four low-end microcontrollers, RL78, ATtiny, Cortex-M0 and MSP430, and demonstrate that slight-looking differences of hardware, in particular carry flag treatment or branch timing, significantly affect size and speed of a target code. We believe that this information is beneficial to not only programmers but also designers of a cryptographic algorithm.

2 How to Minimize AES in Software

In this section we show several techniques to minimize code size of AES. For the specification of AES and the notations, see [7]. Throughout this section, we use C language notations in describing implementation algorithms.

2.1 SubBytes and InvSubBytes

We implement S-box $S(x)$ and the inverse S-box $IS(x)$ as their original formula shown below, using an inversion over $GF(2^8)$, a matrix multiplication on $GF(2)^8$ and an xor of a constant value without any lookup tables.

$$S(x) = \begin{pmatrix} 1&0&0&0&1&1&1&1 \\ 1&1&0&0&0&1&1&1 \\ 1&1&1&0&0&0&1&1 \\ 1&1&1&1&0&0&0&1 \\ 1&1&1&1&1&0&0&0 \\ 0&1&1&1&1&1&0&0 \\ 0&0&1&1&1&1&1&0 \\ 0&0&0&1&1&1&1&1 \end{pmatrix} \cdot x^{-1} + 63, \qquad IS(x) = (x^{-1} + 63) \cdot \begin{pmatrix} 0&0&1&0&0&1&0&1 \\ 1&0&0&1&0&0&1&0 \\ 0&1&0&0&1&0&0&1 \\ 1&0&1&0&0&1&0&0 \\ 0&1&0&1&0&0&1&0 \\ 0&0&1&0&1&0&0&1 \\ 1&0&0&1&0&1&0&0 \\ 0&1&0&0&1&0&1&0 \end{pmatrix}$$

The inversion is an operation on $GF(2)[X]/(X^8 + X^4 + X^3 + X + 1)$ and the matrix multiplication can be regarded as an operation on $GF(2)[X]/(X^8 + 1)$ since it is circular. In other words, a multiplication routine on $GF(2^8)$ can also compute a matrix multiplication on $GF(2)^8$ by just replacing the polynomial.

In general, a (random) matrix calculation is expensive in software, but in our case, the matrix multiplication above becomes almost free by sharing it to the Galois multiplication. This observation leads to the following simple algorithm for computing $S(x)$ and $IS(x)$ as follows:

```
Input x, Output SubBytes(x)
  01:  x = INV8(x)                  : Galois inversion
  02:  x = MUL8(x,0x1f,0x101,0x63) ; matrix multiplication
  03:  return x                     ; (0x101 denotes X^8+1)

Input x, Output InvSubBytes(x)
  04:  x = MUL8(x,0x4a,0x101,0x05) ; matrix multiplication
  05:  x = INV8(x)                  ; Galois inversion
  06:  return x                     ; (0x05 = 0x63 * 0x4a)
```

```
Input x, Output INV8(x)           ; x^254 using a binary method
  07:  c = 0, y = 1
  08:    y = MUL8(y,x,0x11b,0)     ; Galois multiplication
  09:    y = MUL8(y,y,0x11b,0)     ; Galois multiplication
  10:    c = c+1                   ; (0x11b denotes X^8+X^4*X^3+X+1)
  11:    if(c != 7) goto 08
  12:  return y

Input x,y,f,v, Output MUL8(x,y,f,v) ; v=v+(x*y) on GF(2)[X]/(f)
  13:  c = 0
  14:    if((x&1) == 1) v = v^y
  15:    x = x>>1
  16:    y = y<<1
  17:    if(y > 255) y = y^f
  18:    c = c+1
  19:    if(c != 8) goto 14
  20:  return v
```

2.2 AddRoundKey+ShiftRows+SubBytes

`AddRoundKey`, `ShiftRows` and `SubBytes` can be combined into in a very simple loop by noting that `ShiftRows` moves its i-th input byte to the ($i*13$ mod 16)-th output byte ($i = 0, 1, 2, .., 15$). It is easy to see that `InvShiftRows`, `InvSubBytes` and `AddRoundKey` for decryption can be written in a similar way:

```
Input  x[0]..x[15],k[0]..k[15]
Output y[0]..y[15]=(AddRoundKey+ShiftRows+SubBytes)(x,k)
  22:  c = 0, d = 0
  23:    a = x[c]^k[c]       ; AddRoundKey
  24:    y[d] = SubBytes(a)  ; SubBytes (S-box)
  25:    d = d+13 mod 16     ; ShiftRows
  26:    c = c+1
  27:    if(c != 8) goto 23  ; equivalently, if(d == 0) goto 23
  28:  return y
```

2.3 Sharing MixColumns with InvMixColumns

Another trick to minimize AES is to share `MixColumns` with `InvMixColumns` using the following equation, where the middle/left matrix is the one defined in `MixColumns`/`InvMixColumns`, respectively. This decomposition was implicitly used in [6] in the hardware context. Note that all entries in the middle matrix have active bits at bit 0 and/or 1, and all entries in the right matrix have active bits at bit 2 and/or 3.

$$\begin{pmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} + \begin{pmatrix} 0c & 08 & 0c & 08 \\ 08 & 0c & 08 & 0c \\ 0c & 08 & 0c & 08 \\ 08 & 0c & 08 & 0c \end{pmatrix}$$

Using this fact, we can embed `MixColumns` into `InvMixColumns` in the following simple way, where only one additional instruction - a conditional branch - is necessary for detecting encryption/decryption. This algorithm consists of a double loop to minimize its code size, where the inner loop (lines 32 to 44) computes only one vector of the entire four vectors contained in the matrices. We are again using the fact that they are circular.

```
Input  x[0]..x[15]
Output y[0]..y[15]=MixColumns(x) or InvMixColumns(x)
  29:  c0 = 0                               ; matrix number
  30:    a0 = a1 = a2 = a3 = 0
  31:    c1 = 0                             ; vector number
  32:      t = x[c0*4+c1]
  33:      a0 = a0^t, a1 = a1^t, a2 = a2^t  ; 0th bit (ENC)
  34:      t = t*2 on GF(256)
  35:      if ENCRYPTION, then go to 41
  36:        a2 = a2^t, a3 = a3^t           ; 1st bit (DEC)
  37:        t = t*2 on GF(256)
  38:        a1 = a1^t, a3 = a3^t           ; 2nd bit (DEC)
  39:        t = t*2 on GF(256)
  40:        a0 = a0^t, a1 = a1^t           ; 3rd bit (DEC)
  41:      a2 = a2^t, a3 = a3^t             ; 1st/3rd bit (ENC/DEC)
  42:      t = a0, a0 = a1, a2 = a3, a3 = t ; rotate shift
  43:      c1 = c1+1
  44:      if(c1 != 4) goto 32
  45:    y[c0*4] = a0, y[c0*4+1] = a1, y[c0*4+2] = a2, y[c0*4+3] = a3
  46:    c0 = c0+1
  47:    if(c0 != 4) goto 30
  48:  return y
```

Note that [8] reports another decomposition of the `InvMixColumns` matrix as a multiplication of two matrices, not an addition, one of which is the `MixColumns` matrix. Implementing this form in software leads to a bigger code than the above because of the matrix multiplication.

3 Implementation on RL78 and ATtiny

In this section, we discuss our implementations of AES on two low-end microcontrollers, RL78 [9] and ATtiny [10]. RL78 is a typical accumulator-based CISC processor and ATtiny is a typical register-symmetrical RISC processor.

We believe that looking at software implementation on processors with utterly different architecture is of its own interest.

On each processor, we implement three instances: AES encryption-only (AES-E), AES encryption/decryption (AES-ED) and AES Galois counter mode (AES-GCM) [11], based on the algorithms shown in the previous section. Our top priority is to minimize ROM size, and we also try to reduce RAM usage, which is equally important in practice. All codes presented in this section include on-the-fly key scheduling.

3.1 RL78 and ATtiny Microcontrollers

In this subsection we briefly introduce the two microcontrollers, RL78 and ATtiny. More detailed architectural comparison with actual code examples will be discussed in next section.

RL78 has eight 8-bit general-purpose registers `a,x,b,c,d,e,h,l`, of which many instructions accept only `a` or `ax` as a destination. A limited number of instructions have a 16-bit form with register pairs `ax,bc,de,hl`. Unfortunately an xor instruction, frequently used in block ciphers, does not have a 16-bit form [12].

We hence often suffer from "register starvation" on this microcontroller, which requires extra instructions and memory to save/restore data on an accumulator, but a big advantage of RL78 is that code size tends to be short due to its support of read-modify instructions. For instance, `'xor a,[hl]'`, – read from an address pointed by `hl` and xor to `a` –, is a one-cycle instruction with one-byte length. This significantly contributes to code size reduction.

ATtiny has thirty-two 8-bit general-purpose registers `r0` to `r31`. Most instructions have "register symmetry", i.e. accept any register as a destination. Some instructions dealing with immediate data accept only `r16` to `r31`, but this seldom causes trouble to a minimalist. Three register pairs `(r26,r27)`, `(r28,r29)`, and `(r30,r31)` are used as address registers `X`, `Y`, and `Z`, respectively [13].

Almost all instructions of ATtiny are two-byte long and no read-modify instructions are supported. Hence a code size of this RISC microcontroller tends to be bigger than that of CISC RL78, but in general creating a faster code is possible due to less memory accesses and faster jump instructions. The latter is very important because a minimum-size code often consists of many small loops.

Our coding and performance measurement is done on RL78-G12 (ROM 8 K bytes and RAM 768 bytes) and ATtiny85 (ROM 8K bytes and RAM 512 bytes), respectively.

3.2 Interface and Metrics

Defining a software interface clearly is particularly important for discussing a minimal code. For instance, some implementations on ATtiny allow programmers to destroy all registers without restoration [1,2]. Other implementations [14,15] follow the function call conventions described by Atmel [16]. For the latter case, a subroutine code that destroys all registers has to save/restore 18 registers at

the entry and exit of the code, which requires additional 72-byte ROM and 36-byte RAM by pushing/popping these registers. Obviously this overhead is not ignorable in our context.

While our goal is to obtain a size minimum code, we also keep a practical and usable code in mind. We hence adopt the latter approach; that is, we create a function callable from a high-level language and count all resources occupied by a code in referring to ROM/RAM size, as discussed in [3]. The following is our coding and measurement policy in this paper, balancing minimalism, usability and security.

1. Code is described as a subroutine callable from C-language.
2. Code processes one-block data.
3. Code should be relocatable.
4. Execution time is independent of secret information (text and key).
5. ROM size includes instruction code and constant data.
6. RAM size includes plaintext/ciphertext, key, stack, and temporary memory.
7. Plaintext area is shared with ciphertext area.
8. Key area can be destroyed but is recovered at the end of the code.

RAM memory for locating parameters such as text, key and other necessary information, is allocated in consecutive area within a callee code and its address is passed to a caller program as an external variable.

We should clarify policy 2 in the case of AES-GCM. In AES-GCM, we introduce a switch, which we call MODE, that indicates which part of AES-GCM should be carried out, as shown in Fig. 1. Our code reads MODE included in the parameters and executes an appropriate component of the AES-GCM algorithm.

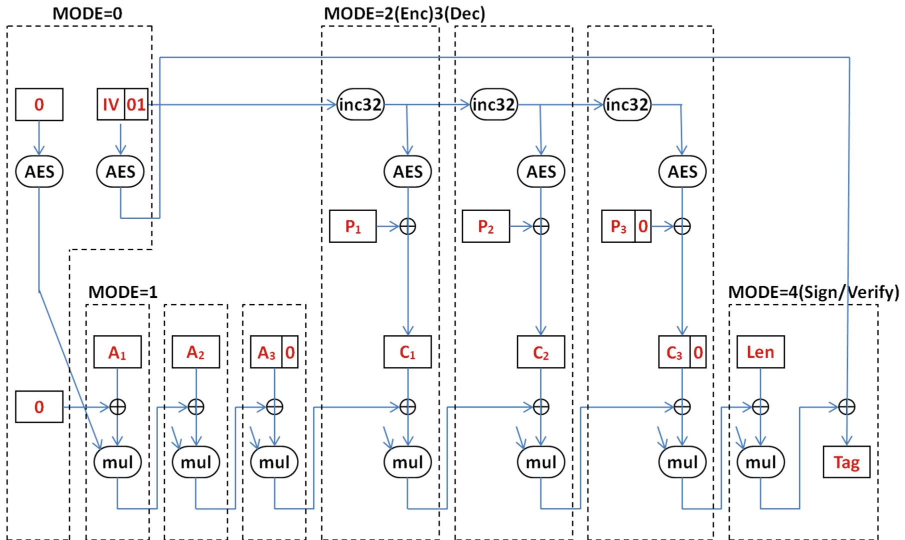

Fig. 1. Our implementation blocks of AES-GCM

Also, strictly speaking we apply policy 3 to our codes of ATtiny in a slightly relaxed way. It is assumed that the RAM memory (excluding stack) resides within the same 256-byte block without crossing a 256-byte aligned boundary. This is because otherwise updating address registers becomes very costly.

3.3 Implementation Results

Table 1 shows our implementation results of AES-E, AES-ED and AES-GCM on RL78 and ATtiny. We followed the coding policy described in the previous subsection and aimed at a minimum memory size. In this table, (E) and (D) denote speed in encryption and decryption, respectively, and (n) in AES-GCM means MODE shown in Fig. 1.

Table 1. Size minimum implementation of AES-E, AES-ED and AES-GCM

Algorithm	Controller	ROM	RAM	Speed (cycles/block)
AES-E	RL78	192	88	369901
AES-E	ATtiny	214	78	262061
AES-ED	RL78	326	103	374016 (E), 449408 (D)
AES-ED	ATtiny	356	78	264302 (E), 318800 (D)
AES-GCM	RL78	429	182	741088 (0), 40620 (1), 412608 (2,3), 40937 (4)
AES-GCM	ATtiny	522	165	525882 (0), 30536 (1), 293506 (2,3), 30876 (4)

We achieved a 192-byte AES encryption code, including on-the-fly key scheduling, on the RL78 microcontroller. It is again noted that it runs in a constant time. As far as we know, this is the smallest AES ever made in software. Of course, its speed is very slow, but still makes sense in non timing-critical applications since it runs in 18.5 ms/block in 20 MHz clock.

For comparison with ATtiny, RL78 is smaller but slower than ATtiny, as expected. More specifically ATtiny is 30 % faster, but 10 % larger for AES-E and AES-ED and 20 % larger for AES-GCM. There are two reasons why ATtiny is much larger in AES-GCM. One is that RL78 has a 16-bit addition instruction and a multiple-bit shift instruction, which are missing in ATtiny. These instructions are efficiently used for counting and accumulating text length in AES-GCM.

Another and more serious reason of this is that ATtiny only allows a 6-bit displacement in register indirect addressing. This means that an address register must be updated every time when it points a new address that is distant from a current address by 64 or more bytes. This restriction does not cause a problem in AES-E and AES-ED since their RAM size is close to 64 bytes, but that of AES-GCM is much larger, which results in visible penalty.

We will discuss more detailed software implementation issues depending on processor hardware architecture in the next section. Here we only illustrate code examples of `AddRoundKey+ShiftRows+SubBytes` to show how a small loop can

be implemented on these microcontrollers in Table 2, where we use the same assembler syntax for RL78 and ATtiny for readers' convenience. It is seen in `AddRoundKey(AR)` that a missing read-modify instruction on ATtiny is compensated by its post increment addressing mode. It is also noted that ATtiny requires more instructions to modify/restore the destination address register at the end of `SubBytes(SB)`, and instead RL78 needs more instructions to do `ShiftRows(SR)` due to its accumulator-based architecture; i.e. no instruction exists such as `'sub c,3'` on RL78.

Table 2. AddRoundKey+ShiftRows+SubBytes on RL78 and ATtiny

```
[RL78: 19byte long]                     [ATtiny: 22byte long]
      clr    c        ; c=0                   clr    r20        ; r20=0
    loop:                                   loop:
(AR)  mov    a,[hl]                     (AR)  mov    r21,[Z+KEY]
(AR)  xor    a,[hl+KEY]                 (AR)  mov    r22,[Z++]
(AR)  inc    hl                         (AR)  xor    r21,r22
(SB)  call   Sbox     ; a->a            (SB)  call   Sbox       ; r21->r0
(SB)  mov    TMP[c],a                   (SB)  add    XL,r20
(SR)  mov    a,c                        (SB)  mov    [X],r0     ; X=TMP+r20
(SR)  sub    a,3                        (SB)  sub    XL,r20
(SR)  and    a,15                       (SR)  sub    r20,3
(SR)  mov    c,a                        (SR)  and    r20,15
(SR)  jnz    loop                       (SR)  jnz    loop
```

3.4 Variations

It is common in a minimum-size approach to see that a slight modification of a source code significantly affects its performance. To see this, we unroll a performance critical loop and measure the size and speed of resultant codes. The performance bottleneck of our AES codes is of course computation of S-box. In particular a multiplication on $GF(2)[X]/(f)$, corresponding to `MUL8` shown in the implementation algorithm of `SubBytes` and `InvSubBytes`, is the critical routine. It consists of a loop with an eight-time iteration (a code example of this routine will be shown in the next section). Unrolling this performance-critical loop improves speed at the cost of a small increase in ROM size as illustrated in Table 3. This table shows that our 520-byte code of AES-GCM on RL78 outperforms the 522-byte code on ATtiny. It should be noted that if we see performance of these microcontrollers with the same ROM size, the lead of ATtiny is not so big.

Table 4 shows another variation of our codes where the S-box and its inversion routines are replaced with normal lookup tables, including previous smallest implementations [3]. We think that ours are still a minimum record of AES, while not a minimalist approach. In the implementation on ATtiny, we put these lookup tables on a 256-byte address boundary for faster memory access, as most implementations of AES on an AVR processor do [1,17,18].

Table 3. Loop unrolled codes of AES-E, AES-ED and AES-GCM on RL78

Algorithm	Controller	#iterations	ROM	RAM	Speed (cycles/blocks)
AES-E	RL78	4	206	88	309901
AES-E	RL78	2	234	88	279901
AES-E	RL78	1	283	88	246901
AES-ED	RL78	4	340	103	314016 (E), 377408 (D)
AES-ED	RL78	2	368	103	284016 (E), 341408 (D)
AES-ED	RL78	1	417	103	251008 (E), 301792 (D)
AES-GCM	RL78	4	442	182	621088 (0), 352608 (2,3)
AES-GCM	RL78	2	471	182	561088 (0), 322592 (2,3)
AES-GCM	RL78	1	520	182	495104 (0), 289600 (2,3)

Table 4. Lookup table implementation of AES-E, AES-ED and AES-GCM

Algorithm	Controller	ROM	RAM	Speed (cycles/block)
AES-E [3]	RL78	486	78	7288
AES-E	RL78	399	78	8704
AES-E	ATtiny	428	82	8870
AES-ED [3]	RL78	970	84	7743 (E), 10362 (D)
AES-ED	RL78	776	85	9847 (E), 13634 (D)
AES-ED	ATtiny	814	82	9624 (E), 13869 (D)
AES-GCM	RL78	642	172	19695 (0), 40640 (1), 51904 (2,3), 40928 (4)
AES-GCM	ATtiny	730	165	19486 (0), 30536 (1), 40308 (2,3), 30876 (4)

In this implementation, ATtiny is 5–10 % larger than RL78 but its speed is comparable with RL78 for AES-E and AES-ED because the performance gain in S-box of ATtiny, which will be demonstrated in the next section, is lost. On the other hand, ATtiny is much faster in all modes of AES-GCM except MODE=0. This is because GHASH of AES-GCM, more specifically a multiplication on $GF(2^{128})$ runs much faster on ATtiny than on RL78. This will be also discussed in the next section.

Lastly, we mention that it is possible to further reduce the code size of our 192-byte program on RL78 by relaxing (or ignoring) the coding policies shown in this section. This is not recommended in general, but might make sense in certain situations. The first possibility is to allow to destroy key data without restoration. Our code copies key to temporary area before starting actual encryption, and hence removing this part reduces code size by 10 bytes. Another possibility is to remove timing-attack protection. In the `MUL8` routine, which is in the bottom of the S-box, we can quit the loop as soon as the shifted multiplier becomes zero, without iterating eight times. The resultant code no longer runs in constant time, but reduces register pressure and saves 4 bytes. Ignoring the

function call convention gains another 2 bytes. Also without violating any policy, x^{254} can be computed by simply multiplying x 253 times instead of using a binary method, which reduces further 4 bytes. We did not adopt this because it makes the code too slow. Applying all these reduces its ROM size down to 172 bytes. It will be a less practical code, though.

4 Minimalism from Hardware Viewpoints

There are various types of microcontrollers currently available in the market, of which low-end ones usually have a similar instruction set consisting of only basic operations such as read/write, arithmetic, logical and branch. However, in practice, minor-looking differences of these instructions often lead to a significant impact on size and speed. This section takes Cortex-M0 [19] and MSP430 [20], in addition to RL78 and ATtiny, as target microcontrollers and demonstrates this fact using concrete code examples for AES. An architectural comparison of these microcontrollers is summarized in the appendix.

Our first example is `MUL8`, a multiplication on $GF(2)[X]/(f)$ used in S-box. The following examples illustrate code minimum implementation of `MUL8` on these four processors. `(1)` corresponds to lines 15 and 16, and `(2)` corresponds to lines 17 and 18 in the sequence shown in Sect. 2.1 (Tables 5 and 6).

Table 5. MUL8 on RL78 (left) and ATtiny (right)

```
[RL78:  b=a*x+b (c=f)]                    [ATtiny: r23=r21*r22+r23 (r24=f)]
[19 bytes, 103 cycles]                    [18 bytes, 72 cycles]
     mov   d,8                                 mov   r25,8
   loop:                                     loop:
(1)  xch   a,x ;exchange a with x         (1)  shr   r22,1
(1)  shr   a,1                            (1)  jnc   L1
(1)  xch   a,x                            (1)  xor   r23,r21
(1)  sknc      ;skip next if non carry      L1:
(1)  xor   b,a
(2)  shl   a,1                            (2)  shl   r21,1
(2)  sknc      ;skip next if non carry    (2)  jnc   L2
(2)  xor   a,c                            (2)  xor   r21,r24
                                             L2:
     dec   d                                   dec   r25
     jnz   loop                                jnz   loop
```

The simplest code is on ATtiny. On RL78, an overhead for creating backup of the accumulator is unavoidable. On the other hand RL78's `sknc` instruction (replace next instruction with `nop` if a condition is met) works fine as a faster alternative of `jnc`. A conditional taken/not-taken jump of Cortex-M0 takes three/one cycles, respectively. This means that a redundant `nop` instruction must be inserted for constant time execution. Also since all registers of

Table 6. MUL8 on Cortex-M0 (left) and MSP430 (right)

```
[Cortex-M0 r6=r4*r5+r6 (r7=f)]
[22 bytes, 102 cycles]
     mov   r1,8
   loop:
(1)  shr   r5,r5,1
(1)  jnc   L1
(1)  xor   r6,r6,r4
(1)  nop   ;protection from
           ;timing attack

   L1:
(2)  shl   r4,r4,1
(2)  shl   r2,r4,24 ;creating carry
(2)  jnc   L2
(2)  xor   r4,r4,r7
(2)  nop   ;protection from
           ;timing attack
   L2:
     sub   r1,r1,1
     jnz   loop
```

```
[MSP430 r12=r11*r10+r12 (r13=f)]
[26 bytes, 121 cycles]
     mov   r14,8
   loop:
(1)  shr   r11,1
(1)  jc    L1a ;protection from
(1)  nop       ;timing attack
  L1a:
(1)  jnc   L1b
(1)  xor   r12,r10
   L1b:
(2)  shl   r10,1
(2)  jc    L2a ;protection from
(2)  nop       ;timing attack
   L2a:
(2)  jnc   L2b
(2)  xor   r10,r13
   L2b:
     sub   r14,1
     jnz   loop
```

Cortex-M0 are 32-bit long only, an extra instruction is required to create the carry flag. Interestingly, a conditional jump of MSP430 always takes two cycles, and hence a special care is need to create a timing-attack protected code. To do this we insert a dummy conditional jump instruction with an opposite logic for each branch, which causes a heavy size and performance penalty.

The next example is a multiplication on $GF(2^{128})$ that appears in GHASH of AES-GCM. The following codes show part of one iteration of the multiplication. More specifically, the code consists of two functions: (1) If carry (or the highest bit of a register) is active, then $A = A$ xor B, (2) Rotate right shift B by one bit. A and B are 128-bit data pointed by an address register.

The most straightforward code is RL78. Note that in the first loop the carry flag must be checked every time to make the code run in constant time. Since both loops handle the carry flag independently, it is not trivial to combine them into a single loop. However for ATtiny, thanks to its `sbrc` instruction (replace next instruction with `nop` if a bit on a register is non active), this can be done in a very simple way (Table 7).

An obstacle of Cortex-M0 and MSP430 is that they do not have a decrement instruction that does not touch the carry flag. In general carry-free `dec/inc` instructions are frequently used for arithmetic of long integers. Moreover Cortex-M0 does not have a rotate-shift-with-carry instruction. Hence we have to create a rather tricky code to simulate it. This causes a significant penalty. MSP430 again suffers a speed overhead for timing adjustment, while the code is very simply described due to its abundant addressing mode.

Table 7. Multiplication on $GF(2^{128})$ on RL78 (left) and ATtiny (right)

```
[RL78]
[23 bytes, 302 cycles]
     mov   bc,#1010h
   loop1:
     dec   hl
(1)  mov   a,[hl+disp]
(1)  sknc  ;check carry
(1)  xor   a,[hl]
(1)  mov   [hl+disp],a
     dec   b
     jnz   loop1

     cmp0  a ;reset carry
   loop2:
(2)  mov   a,[hl]
(2)  rorc  a,1 ;rotation with carry
(2)  mov   [hl],a
     inc   hl
     dec   c
     jnz   loop2
```

```
[ATtiny]
[22 bytes, 225 cycles]
     clc            ;reset carry
     mov   r22,16
   loop1:
(1)  mov   r20,[Z]
(1)  mov   r21,[Z+disp]
(1)  sbrc  r23,7    ;skip next if 7th bit is 0
(1)  xor   r20,r21
(2)  rorc  r21,1    ;rotation with carry
(2)  mov   [Z+disp],r21
(1)  mov   [Z++],r20
     dec   r22
     jnz   loop1
```

Table 8. Multiplication on $GF(2^{128})$ on Cortex-M0 (left) and MSP430 (right)

```
[Cortex-M0]
[32 bytes, 320 cycles]
     mov   r1,0      ;reset carry
     mov   r4,16
   loop:
(1)  mov   r6,[r7]
(1)  shl   r5,r2,25 ;check 7th bit
(1)  mov   r5,[r7+16]
(1)  xor   r6,r6,r5
(1)  jnc   L1
(1)  mov   [r7],r6
   L1:
(2)  shl   r1,r1,7  ;r1=00 or 80
(2)  shr   r5,r5,1
(2)  xor   r5,r5,r1
(2)  mov   [r7+16],r5
(2)  adc   r1,r1,r1 ;r1=00 or 01
     add   r7,r7,1
     sub   r4,r4,1
     jnz   loop
```

```
[MSP430]
[30 bytes 291 cycles]
     mov   r13,0            ;reset carry
     mov   r14,16
   loop:
(1)  test  r12              ;check 7th bit
(1)  jl    L1               ;protection from
(1)  xor   [r11+0],[r11]    ;timing attack
   L1:
(1)  jge   L2
(1)  xor   [r11+16],[r11]
   L2:
(2)  rorc  r13,1            ;save carry
(2)  rorc  [r11++],1
(2)  rolc  r13,1            ;restore carry
     sub   r14,1
     jnz   loop
```

For Cortex-M0 and MSP430, we can write a much faster code by fully using their 32-bit/16-bit wide registers. However this results in an increase in code size because we need extra byte-swap instructions due to little-endianness of these microcontrollers (Table 8).

5 Concluding Remarks

In this paper we explored minimalism in software implementation of AES on various modern low-end microcontrollers. As far as the authors know, this is the first extensive analysis of embedded software coding of symmetric primitives toward memory size reduction with comparative viewpoints of processor hardware. As concluding remarks, we mention some lessons we learned which could be beneficial to programmers and designers of symmetric primitives for low-end microcontrollers.

Use left shifts. Availability and efficiency of shift instructions greatly depends on processor hardware. Some do not support shift with carry instructions. Then `adc` (addition with carry) instruction can be an alternative of a left rotation with carry.

Be aware of locality of RAM access. ATtiny accepts only 6-bit displacement in its register indirect addressing, which is often restrictive. An order of parameters such as text, key, temporary subkey etc., can affect code size and performance.

Why not little endian. Most modern processors have a little endian hardware, but most modern symmetric encryption algorithms are suitable to a big endian architecture. RL78, ATtiny, Cortex-M0, MSP430 are all little endian.

Matrix should be circular. A circular matrix significantly contributes to code reduction. First of all, a matrix multiplication with a circular matrix can be described using a vector-wise loop, and also can be regarded as a multiplication on a ring, as shown in this paper.

Appendix: Low End Microcontrollers Comparison Chart

This table is not intended to be exhaustive, but to illustrate typical cases in implementing lightweight symmetric ciphers for readers' convenience.

	RL78	ATtiny	CortexM0	MSP430
Hardware Registers				
- Register Size	8,16	8	32	8,16
- Number of General Registers	8	32	13	12
Addressing Modes				
- Number of Operands	2	2	2,3	2
- Read-Modify(-Write) Instructions	R-M	No	No	R-M-W
- Post-Increment Addressing	No	Yes	No	Yes
Code Length (bytes)				
- Operation equivalent to `xor reg,[mem]`	1-3	4	4	2,4
- Conditional Short Jump	2	2	2	2
- Subroutine Call	3	2	4	4
Instruction Latency (cycles)				
- Read from Memory (RAM/ROM)	1/4	2/3	2	2
- Operation equivalent to `xor reg,[mem]`	1	2	2	2-3
- Conditional Short Jump (taken/not-taken)	4/2	2/1	3/1	2/2
- Call+Return	9	7	7	7
Supported Instructions				
- Shift with multiple counts	Yes	No	Yes	No
- Rotate Shift without carry	Yes	No	Yes	No
- Rotate Shift with carry	Yes	Yes	No	Yes
- Carry preserving increment/decrement	Yes	Yes	No	No
- Conditional Skip	Yes	Yes	No	No

References

1. Eisenbarth, T., et al.: Compact implementation and performance evaluation of block ciphers in ATtiny devices. In: Mitrokotsa, A., Vaudenay, S. (eds.) AFRICACRYPT 2012. LNCS, vol. 7374, pp. 172–187. Springer, Heidelberg (2012)
2. Balasch, J., et al.: Compact implementation and performance evaluation of hash functions in ATtiny devices. In: Mangard, S. (ed.) CARDIS 2012. LNCS, vol. 7771, pp. 158–172. Springer, Heidelberg (2013). http://eprint.iacr.org/2012/507.pdf
3. Matsui, M., Murakami, Y.: Minimalism of software implementation-extensive performance analysis of symmetric primitives on the RL78 microcontroller. In: Moriai, S. (ed.) FSE 2013. LNCS, vol. 8424, pp. 393–409. Springer, Heidelberg (2014)
4. Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., Wingers, L.: The SIMON and SPECK Families of Lightweight Block Ciphers. http://eprint.iacr.org/2013/404.pdf
5. Papagiannopoulos, K., Verstegen, A.: Speed and size-optimized implementations of the PRESENT cipher for tiny AVR devices. In: Hutter, M., Schmidt, J.-M. (eds.) RFIDsec 2013. LNCS, vol. 8262, pp. 161–175. Springer, Heidelberg (2013)

6. Fischer, V., Drutarovsky, M., Chodowiec, P., Gramain, F.: InvMixColumn decomposition and multilevel resource sharing in AES implementations. IEEE Trans. VLSI Syst. **13**(8), 989–992 (2005)
7. Advanced Encryption Standard (AES), Federal Information Processing Standards Publication 197, NIST (2001)
8. Daemen, J., Rijmen, V.: The Design of Rijndael. Springer, Heidelberg (2002)
9. Renesas Electronics, RL78 Family. http://am.renesas.com/products/mpumcu/rl78/index.jsp?campaign=gn_prod
10. Atmel, tinyAVR Microcontrollers. http://www.atmel.com/products/microcontrollers/avr/tinyavr.aspx
11. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC, Special Publication 800–38D, NIST (2007)
12. RL78 Family, User's Manual. http://documentation.renesas.com/doc/products/mpumcu/doc/rl78/r01us0015ej0210_rl78.pdf
13. 8-bit AVR Instruction Set http://www.atmel.com/Images/doc0856.pdf
14. AVR-Crypto-Lib Wiki. http://www.das-labor.org/wiki/AVR-Crypto-Lib/en
15. Gauravaram, P., Knudsen, L.R., Matusiewicz, K., Mendel, F., Rechberger, C., Schläffer, M., Thomsen, S.S.: Grøstl - a SHA-3 candidate. http://www.groestl.info/
16. Mixing Assembly and C with AVRGCC. http://www.atmel.com/Images/doc42055.pdf
17. Bos, J.W., Osvik, D.A., Stefan, D.: Fast Implementations of AES on Various Platforms. http://eprint.iacr.org/2009/501.pdf
18. Poettering, B.: Rijndael Furious. http://perso.uclouvain.be/fstandae/lightweight_ciphers/source/AES_furious.asm
19. ARM Cortex-M0 core MCUs. http://www.nxp.com/products/microcontrollers/cortex_m0_m0/
20. Overview for MSP430 Ultra-Low Power 16-bit MCUs. http://www.ti.com/lsds/ti/microcontroller/16-bit_msp430/overview.page

Attacks

Differential Factors: Improved Attacks on SERPENT

Cihangir Tezcan[1,2](✉) and Ferruh Özbudak[1]

[1] Department of Mathematics and Institute of Applied Mathematics, Middle East Technical University, Ankara, Turkey

[2] Institute of Informatics, CyDeS Cyber Defence and Security Lab, Middle East Technical University, Ankara, Turkey

cihangir@metu.edu.tr

Abstract. A differential attack tries to capture the round keys corresponding to the S-boxes activated by a differential. In this work, we show that for a fixed output difference of an S-box, it may not be possible to distinguish the guessed keys that have a specific difference. We introduce these differences as differential factors. Existence of differential factors can reduce the time complexity of differential attacks and as an example we show that the 10, 11, and 12-round differential-linear attacks of Dunkelman *et al.* on SERPENT can actually be performed with time complexities reduced by a factor of 4, 4, and 8, respectively.

Keywords: S-box · Differential factor · SERPENT · Differential-linear attack

1 Introduction

Confusion layer of cryptographic algorithms mostly consists of substitution boxes (S-boxes) and in order to provide better security against known attacks, S-boxes are selected depending on their cryptographic properties: Low non-linear and differential uniformity [30] provide resistance against linear and differential cryptanalysis, respectively; high algebraic degree and branch number provides resistance against algebraic [14] and cube [16] attacks; lack of undisturbed bits [37] provides resistance against truncated [20], impossible [2], and improbable [36] differential cryptanalysis. Moreover, recently it was shown in [9] that additive shares can be used in threshold implementations to provide resistance against side-channel attacks like differential power analysis [21] and the number of shares affects the performance.

In this work, we show that a fixed S-box output difference μ may remain invariant when the possible input pairs are XORed with some λ. We define such λ as a differential factor for the output difference μ and we show that such a

C. Tezcan—The work of the first author was supported by The Scientific and Technological Research Council of Turkey (TÜBİTAK) under the grant 112E101 titled "Improbable Differential Cryptanalysis of Block Ciphers".

© Springer International Publishing Switzerland 2015

T. Eisenbarth and E. Öztürk (Eds.): LightSec 2014, LNCS 8898, pp. 69–84, 2015.
DOI: 10.1007/978-3-319-16363-5_5

property of an S-box can significantly reduce the attacked key space in a differential attack which results in an attack with reduced time complexity. Our analysis of S-boxes that are used in cryptographic algorithms show that differential factors are observed mostly in small S-boxes. We observed that 73 % of all possible bijective 3×3 S-boxes contain differential factors. Moreover, 4×4 S-boxes of DES [28], GOST [39], LBLOCK [41], LED [18], LUFFA [12], NOEKEON [15], *Piccolo* [35], PRESENT [11], RECTANGLE [42], SARMAL [40], SERPENT [1], SPONGENT [10] and Twofish [33] contain differential factors.

Lightweight cryptography has become very vital with the emerging needs in sensitive applications like RFID (Radio-frequency identification) systems and sensor networks. For these types of special purposes, there is a strong demand in designing secure lightweight cryptographic modules. Since most of such lightweight algorithms have a hardware oriented design, they use small S-boxes. Thus, differential factors pose a threat to lightweight block ciphers. We indicate the importance of this new S-box criteria on cryptanalysis by reducing the time complexities of the 10, 11, and 12-round differential-linear attacks of Dunkelman *et al.* on SERPENT by a factor of 4, 4, and 8, respectively. By changing the differential, we further modify these attacks to marginally reduce the data complexity. We compare our improved attacks on SERPENT with the previous ones in Table 1.

2 S-Box Evaluation

S-boxes are commonly used as non-linear components for symmetric cryptosystems and hash functions. Properties of S-boxes provide resistance against many cryptanalytic techniques.

Differential Uniformity

Definition 1. *For a mapping $S : F_2^n \to \mathbb{F}_2^m$, and all $\Delta_i \in \mathbb{F}_2^n$ and $\Delta_o \in \mathbb{F}_2^m$, let t be the number of elements x that satisfy $S(x \oplus \Delta_i) = S(x) \oplus \Delta_o$. Then $t/|2^n|$ is the differential probability of the characteristic $S(\Delta_i \to \Delta_o)$. The table that lists all t values for every $i, o \in X$ is called the* Difference Distribution Table (DDT).

The maximum value in a DDT, excluding the zero difference case, is called differential uniformity. S-box designers aim to minimize differential uniformity since differential cryptanalysis [8] uses characteristics with high differential probability.

Non-linear Uniformity

Definition 2. *For a mapping $S : \mathbb{F}_2^n \to \mathbb{F}_2^m$, and all $a \in \mathbb{F}_2^n$ and $b \in \mathbb{F}_2^m$, let the numbers $L_f(a, b)$ be defined as*

$$L_f(a, b) := |\#\{x \in F_2^n | a \cdot x = b \cdot S(x)\} - 2^{n-1}|$$

where $a \cdot b$ denotes the parity of the bit-wise product of a and b. Then S is called non-linearly l-uniform *if $L_f(a, b) \le l$ for all a and b with $b \neq 0$.*

Table 1. Summary of attacks on SERPENT. Note that it is claimed in [27] that the multidimensional linear attacks of [29] may not work as claimed depending on the linear hull effect. If the claims are correct, then our use of differential factors in the attacks of [17] becomes the best attacks for this cipher.
En - Encryptions, MA - Memory Accesses, B - bytes, CP - Chosen Plaintexts, KP - Known Plaintexts.

#Rounds	Attack Type	Key Size	Data	Time	Memory	Advantage	Success	Reference
6	Meet-in-the-middle	256	512 KP	2^{247} En	2^{246} B	-	-	[22]
6	Differential	All	2^{83} CP	2^{90} En	2^{40} B	-	-	[22]
6	Differential	All	2^{71} CP	2^{103} En	2^{75} B	-	-	[22]
6	Differential	192, 256	2^{41} CP	2^{163} En	2^{45} B	124	-	[22]
7	Differential	256	2^{122} CP	2^{248} En	2^{126} B	128	-	[22]
7	Improbable	All	$2^{116.85}$ CP	$2^{117.57}$ En	2^{113} B	112	99.9%	[38]
7	Differential	All	2^{84} CP	2^{85} MA	2^{56} B	-	-	[4]
10	Rectangle	192, 256	$2^{126.3}$ CP	$2^{173.8}$ MA	$2^{131.8}$ B	80	-	[6]
10	Boomerang	192, 256	$2^{126.3}$ AC	$2^{173.8}$ MA	2^{89} B	80	-	[6]
10	Differential-Linear	All	$2^{101.2}$ CP	$2^{115.2}$ En	2^{40} B	40	84%	[17]
10	**Differential-Linear**	**All**	$\mathbf{2^{101.2}}$ **CP**	$\mathbf{2^{113.2}}$ **En**	$\mathbf{2^{40}}$ **B**	**38**	**84%**	**Sect. 4.4**
10	**Differential-Linear**	**All**	$\mathbf{2^{100.55}}$ **CP**	$\mathbf{2^{116.55}}$ **En**	$\mathbf{2^{40}}$ **B**	**42**	**84%**	**Appx. B**
11	Linear	256	2^{118} KP	2^{214} MA	2^{85} B	140	78.5%	[3]
11	Multidimensional Linear [a]	All	2^{116} KP	$2^{107.5}$ En	2^{108} B	48	78.5%	[29]
11	Multidimensional Linear [b]	All	2^{118} KP	$2^{109.5}$ En	2^{104} B	44	78.5%	[29]
11	Nonlinear	192, 256	$2^{120.36}$ KP	$2^{139.63}$ MA	$2^{133.17}$ B	118	78.5%	[27]
11	Filtered Nonlinear	192, 256	$2^{114.55}$ KP	$2^{155.76}$ MA	$2^{146.59}$ B	132	78.5%	[27]
11	Differential-Linear	192, 256	$2^{121.8}$ CP	$2^{135.7}$ MA	2^{76} B	48	84%	[17]
11	**Differential-Linear**	**192, 256**	$\mathbf{2^{121.8}}$ **CP**	$\mathbf{2^{133.7}}$ **MA**	$\mathbf{2^{76}}$ **B**	**46**	**84%**	**Sect. 4.4**
11	**Differential-Linear**	**192, 256**	$\mathbf{2^{121.15}}$ **CP**	$\mathbf{2^{137.05}}$ **MA**	$\mathbf{2^{76}}$ **B**	**50**	**84%**	**Appx. B**
12	Multidimensional Linear [c]	256	2^{116} KP	$2^{237.5}$ En	2^{125} B	174	78.5%	[29]
12	Differential-Linear	256	$2^{123.5}$ CP	$2^{249.4}$ En	$2^{128.5}$ B	160	84%	[17]
12	**Differential-Linear**	**256**	$\mathbf{2^{123.5}}$ **CP**	$\mathbf{2^{246.4}}$ **En**	$\mathbf{2^{128.5}}$ **B**	**157**	**84%**	**Sect. 4.4**

[a] In [27], it is claimed that the correct data complexity of this attack is $2^{125.81}$ KP and the time complexity is $2^{101.44}$ En $+2^{114.13}$ MA.
[b] In [27], it is claimed that the correct data complexity of this attack is $2^{127.78}$ KP and the time complexity is $2^{97.41}$ En $+2^{110.10}$ MA.
[c] In [27], it is claimed that the correct data complexity of this attack is $\geq 2^{125.81}$ KP and the time complexity is $2^{229.44}$ En $+2^{242.13}$ MA.

S-box designers aim to minimize the non-linear uniformity l since linear cryptanalysis [26] uses linear approximations with high bias.

Branch Number

Definition 3. *[32] The branch number of an $n \times n$ S-box is*

$$BN = \min_{a,b\neq a}(wt(a \oplus b) + wt(S(a) \oplus S(b))),$$

where $a, b \in X$ and $wt(a)$ is the Hamming weight of the bit vector a.

For a bijective S-box, the branch number is at least 2 and this property of S-boxes is closely related to algebraic [14] and cube attacks [16].

Number of Shares. S-boxes are also studied for their security against side-channel attacks. Side-channel attacks are based on the information leakage during the computation of the hardware implementation of a cryptographic algorithm. For instance, differential power analysis (DPA) [21] exploits the correlation between the instantaneous power consumption of a device and the intermediate results of a cryptographic algorithm. One countermeasure against side-channel attacks is threshold implementation in which a variable is split into additive shares. Bilgin *et al.* analyzed the number of shares of S-boxes by categorizing all 3×3 and 4×4 S-boxes using affine equivalence classes and investigated the cost of this kind of protection in [9].

Undisturbed Bits. Recently in [37], undisturbed bits are introduced as probability 1 truncated differentials for S-boxes. A 13-round improbable differential attack on Present that uses undisturbed bits is provided in [37] and it was shown that the attack reduces to 7 rounds when the S-box is replaced with a similar one that lacks undisturbed bits. Moreover, it is shown that every bijective 3×3 S-box contains undisturbed bits and a list of ciphers were provided in [37] whose 4×4 S-boxes contain undisturbed bits. S-boxes with undisturbed bits should be avoided to increase security against truncated, impossible, and improbable differential cryptanalysis.

3 Differential Factors

A differential attack on block ciphers tries to capture the round keys corresponding to the S-boxes activated by a differential. However, output difference of the S-box operation may be invariant when the round key is XORed with some specific value. Such a case would prevent the attacker from fully capturing the round key. This observation is similar to the *linear factors* of block ciphers but here we are focusing on the S-box instead of some rounds of the cipher and we focus on key differences.

Definition 4 ([13]). *A block cipher is said to have a* linear factor *if, for all plaintexts and keys, there is a fixed non-empty set of key bits whose simultaneous complementation leaves the XOR sum of a fixed non-empty set of ciphertext bits unchanged.*

In order to have a similar property for S-boxes in the concept of differential cryptanalysis, we define the differential factors as follows:

Definition 5. *Let S be a function from $\mathbb{F}_2^n$ to $\mathbb{F}_2^m$. For all $x, y \in \mathbb{F}_2^n$ that satisfy $S(x) \oplus S(y) = \mu$, if we also have $S(x \oplus \lambda) \oplus S(y \oplus \lambda) = \mu$, then we say that the S-box has a* differential factor *λ for the output difference μ. (i.e. μ remains invariant for λ).*

When undisturbed bits are introduced in [37], the undisturbed bits of S-boxes and their inverses are considered together because in substitution permutation networks (SPNs), the inverse of an S-box is used for decryption. For instance, a 6-round impossible differential for Present is obtained in [37] by using both

undisturbed bits of its S-box and the inverse of it. In the following theorem, we prove that the number of differential factors of an S-box is the same with the number of differential factors of its inverse. Moreover, it also provides the differential factors of the inverse S-box when we know the differential factors of the S-box. Hence, there is no need to check the differential factors of the inverse of S-boxes.

Theorem 1. *If a bijective S-box S has a differential factor λ for an output difference μ, then S^{-1} has a differential factor μ for the output difference λ.*

Proof. Let us assume that S has a differential factor λ for an output difference μ. If $S^{-1}(c_1) \oplus S^{-1}(c_2) = \lambda$ for some c_1 and c_2, then we need to show that $S^{-1}(c_1 \oplus \mu) \oplus S^{-1}(c_2 \oplus \mu) = \lambda$.

Let $c_1 \oplus \mu = S(p_1)$ for some p_1, then we have $S(S^{-1}(c_1) \oplus \lambda) \oplus S(p_1 \oplus \lambda) = \mu$ since λ is a differential factor of S for μ. Thus, we have

$$\begin{aligned} S^{-1}(c_1 \oplus \mu) \oplus S^{-1}(c_2 \oplus \mu) &= S^{-1}(S(p_1)) \oplus S^{-1}(S(S^{-1}(c_1) \oplus \lambda) \oplus \mu) \\ &= p_1 \oplus S^{-1}(S(p_1 \oplus \lambda)) \\ &= p_1 \oplus p_1 \oplus \lambda \\ &= \lambda \end{aligned}$$

□

Theorem 2. *If λ_1 and λ_2 are differential factors for an output difference μ, then $\lambda_1 \oplus \lambda_2$ is also a differential factor for the output difference μ. i.e. All differential factors λ_i for μ form a vector space.*

Proof. We are going to use the following change of variables: $x' = x \oplus \lambda_1$ and $y' = y \oplus \lambda_1$. For all (x, y) pairs satisfying $S(x) \oplus S(y) = \mu$, we have $S(x \oplus \lambda_1) \oplus S(y \oplus \lambda_1) = \mu$ and $S(x \oplus \lambda_2) \oplus S(y \oplus \lambda_2) = \mu$. Thus, we have

$$S(x \oplus \lambda_1 \oplus \lambda_2) \oplus S(y \oplus \lambda_1 \oplus \lambda_2) = S(x' \oplus \lambda_2) \oplus S(y' \oplus \lambda_2) = \mu$$

□

In this section we used two variables x and y since they are directly linked to the input pairs in differential cryptanalysis. However, same definition and theorems can be given using a single variable for bijective S-boxes and we provide them in Appendix A.

3.1 Differential Factors and Cryptanalysis

We start by recalling the definition of advantage.

Definition 6 ([34]). *If an attack on an m-bit key gets the correct value ranked among the top r out of 2^m possible candidates, we say the attack obtained an $(m - log(r))$-bit* advantage *over exhaustive search.*

Theorem 3. *In a block cipher let an S-box S contain a differential factor λ for an output difference μ and the partial round key k is XORed with the input of S. If an input pair provides the output difference μ under a partial subkey k, then*

the same output difference is observed under the partial subkey $k \oplus \lambda$. Therefore, during a differential attack involving the guess of a partial subkey corresponding to the output difference μ, the advantage of the cryptanalyst is reduced by 1 *bit and the time complexity of this key guess step is halved.*

Proof. In a differential attack for any key k, k and $k \oplus \lambda$ would get the same number of hits since λ is a differential factor. Hence the attacker cannot distinguish half of the guessed keys with the other half. Therefore during the key guessing step, the attacker does not need to guess half of the keys. Thus, the time complexity of this step is halved. □

Corollary 1. *During a differential attack involving the guess of a partial subkey corresponding to the output difference μ of an S-box that has a vector space of differential factors of dimension r for μ, the advantage of the cryptanalyst is reduced by r bits and the time complexity of the key guess step is reduced by a factor of 2^r.*

Proof. Follows directly from Theorems 2 and 3. □

3.2 Relating Differential Factors to Other Properties of S-Boxes

Since we are considering non-zero μ and λ, a 3×3 S-box can contain at most $7 \cdot 7 = 49$ differential factors. In such a case, an S-box provides no security at all. In [37], it was shown that every bijective 3×3 S-box contains undisturbed bits. However, this is not the case for differential factors. Among the $8! = 40320$ different bijective 3×3 S-boxes, we observed that 10752 of them do not contain any differential factor. Moreover, 18816 of them contain 9, 9408 of them contain 25, and 1344 of them contain 49 differential factors.

We further observed that the 3×3 S-boxes that do not have any differential factor also have 6 undisturbed bits, which is the smallest number of undisturbed bits a 3×3 S-box can have. Thus, for the case of 3×3 S-boxes, it is enough to check differential factors.

In our literature search we found 102 unique 4×4 S-boxes that are used in block ciphers and hash functions and observed that 40 of them have 74 differential factors in total, without counting the differential factors of their inverses. These are the S-boxes of DES, GOST, LBLOCK, LED, LUFFA, NOEKEON, *Piccolo*, PRESENT, RECTANGLE, SARMAL, SERPENT, SPONGENT and Twofish and they are provided in Table 2.

During our analysis, we observed that the existence of differential factors for an S-box is closely related to the number of nonzero entries in the columns of the DDT table. For instance, for a differentially 4-uniform 4×4 S-box, which is the best case for S-boxes of this size, we observed the following phenomenon:

Conjecture 1. A differential 4-uniform 4×4 S-box S has a differential factor for the output difference μ if and only if the μ-th column of the DDT table of S consists of only zeros and fours.

The only 8×8 S-boxes we found with differential factors are the two S-boxes of the initial version of the CRYPTON cipher [24]. They contain 15 differential factors each and they are provided in Table 2. These S-boxes are replaced in the revised version of the CRYPTON cipher [25] and the new S-boxes do not contain any differential factors.

4 Improved Differential-Linear Attacks on SERPENT

4.1 SERPENT

SERPENT was designed by Anderson, Biham and Knudsen in 1998. It was submitted to the AES contest and came second after Rijndael. It has a block size of 128 bits and accepts any key size of length 0 to 256 bits. It is a 32-round SPN, where each round consists of key mixing, a layer of S-boxes and a linear transformation.

The 128-bit input value before round i is denoted by $\hat{B}_i$, $i \in \{0, \ldots, 31\}$. Each $\hat{B}_i$ is composed of four 32-bit words X_0, X_1, X_2, X_3 where X_0 is the left-most word.

Three round operations are specified as follows:

1. Key Mixing: At each round R_i, a 128-bit subkey K_i is XORed with the current intermediate data $\hat{B}_i$.
2. S-boxes: At each round, R_i uses a single S-box S_j, where $i \equiv j \pmod 8$ and $i \in \{0, \ldots, 31\}$, 32 times in parallel. In this paper, we use the bitsliced version of SERPENT. For example, in the first round the first copy of S_0 takes the least significant bits from X_0, X_1, X_2, X_3 and returns the output to the same bits. Thus, we obtain 32 4-bit slices referred as b_i's, where $i \in \{0, \ldots, 31\}$ and b_0 is the right most slice.
3. Linear Transformation: The four 32-bit words X_0, X_1, X_2, X_3 are linearly mixed by the following linear operations:

$$
\begin{aligned}
X_0 &:= X_0 \lll 13 \\
X_2 &:= X_2 \lll 3 \\
X_1 &:= X_1 \oplus X_0 \oplus X_2 \\
X_3 &:= X_3 \oplus X_2 \oplus (X_0 \ll 3) \\
X_1 &:= X_1 \lll 1 \\
X_3 &:= X_3 \lll 7 \\
X_0 &:= X_0 \oplus X_1 \oplus X_3 \\
X_2 &:= X_2 \oplus X_3 \oplus (X_1 \ll 7) \\
X_0 &:= X_0 \lll 5 \\
X_2 &:= X_2 \lll 22 \\
\hat{B}_{i+1} &:= X_0, X_1, X_2, X_3
\end{aligned}
$$

where $\lll$ denotes the left rotation operation and $\ll$ denotes the left shift operation.

Table 2. Differential Factors of Cryptographic Algorithms

S-box	λ	μ
CRYPTON S_0, S_1	10_x	10_x
CRYPTON S_0, S_1	20_x	20_x
CRYPTON S_0, S_1	30_x	30_x
CRYPTON S_0, S_1	40_x	40_x
CRYPTON S_0, S_1	50_x	50_x
CRYPTON S_0, S_1	60_x	60_x
CRYPTON S_0, S_1	70_x	70_x
CRYPTON S_0, S_1	80_x	80_x
CRYPTON S_0, S_1	90_x	90_x
CRYPTON S_0, S_1	$A0_x$	$A0_x$
CRYPTON S_0, S_1	$B0_x$	$B0_x$
CRYPTON S_0, S_1	$C0_x$	$C0_x$
CRYPTON S_0, S_1	$D0_x$	$D0_x$
CRYPTON S_0, S_1	$E0_x$	$E0_x$
CRYPTON S_0, S_1	$F0_x$	$F0_x$
DES1 Row3	F_x	2_x
DES1 Row3	F_x	8_x
DES1 Row3	F_x	A_x
DES2 Row1	6_x	A_x
DES2 Row2	2_x	7_x
DES2 Row2	4_x	7_x
DES2 Row2	6_x	7_x
DES2 Row3	1_x	A_x
DES2 Row3	6_x	A_x
DES2 Row3	7_x	A_x
DES3 Row3	2_x	6_x
DES3 Row3	8_x	6_x
DES3 Row3	A_x	6_x
DES3 Row4	3_x	1_x
DES3 Row4	3_x	6_x
DES3 Row4	3_x	7_x
DES3 Row4	3_x	8_x
DES3 Row4	3_x	9_x
DES3 Row4	1_x	E_x
DES3 Row4	2_x	E_x
DES3 Row4	3_x	E_x
DES3 Row4	3_x	F_x
DES5 Row4	2_x	F_x
DES6 Row1	9_x	D_x
DES6 Row2	B_x	4_x
DES6 Row4	6_x	6_x
DES7 Row2	4_x	D_x
DES7 Row2	9_x	D_x
DES7 Row2	D_x	D_x
DES7 Row4	4_x	3_x

S-box	λ	μ
DES7 Row4	1_x	C_x
DES7 Row4	4_x	C_x
DES7 Row4	5_x	C_x
DES7 Row4	4_x	F_x
DES8 Row2	6_x	7_x
DES8 Row2	B_x	8_x
GOST S_1	5_x	3_x
GOST S_4	D_x	5_x
GOST S_6	9_x	B_x
GOST S_8	7_x	5_x
GOST S_8	E_x	6_x
LBLOCK S_0, S_8	B_x	1_x
LBLOCK S_0, S_8	3_x	4_x
LBLOCK S_1, S_6, S_7, S_9	B_x	2_x
LBLOCK S_1, S_6, S_7, S_9	3_x	4_x
LBLOCK S_2	3_x	1_x
LBLOCK S_2	B_x	2_x
LBLOCK S_3	B_x	1_x
LBLOCK S_3	3_x	8_x
LBLOCK S_4, S_5	B_x	1_x
LBLOCK S_4, S_5	3_x	2_x
LUFFA	4_x	1_x
LUFFA	2_x	2_x
NOEKEON	1_x	1_x
NOEKEON	B_x	B_x
Piccolo	1_x	2_x
Piccolo	2_x	5_x
PRESENT, LED	1_x	5_x
PRESENT, LED	F_x	F_x
RECTANGLE	2_x	4_x
RECTANGLE	E_x	C_x
SARMAL S_2	F_x	4_x
SARMAL S_2	A_x	9_x
SERPENT S_0	4_x	4_x
SERPENT S_0	D_x	F_x
SERPENT S_1	4_x	4_x
SERPENT S_1	F_x	E_x
SERPENT S_2	2_x	1_x
SERPENT S_2	4_x	D_x
SERPENT S_6	6_x	2_x
SERPENT S_6	F_x	F_x
SPONGENT	F_x	9_x
SPONGENT	1_x	F_x
Twofish q0 t1	6_x	9_x
Twofish q1 t2	5_x	B_x

32-round Serpent cipher may be described by the following equations:

$$\hat{B}_0 := P \qquad \hat{B}_{i+1} := R_i(\hat{B}_i),\ i \in \{0, \ldots, 31\} \qquad C := \hat{B}_{32}$$

where

$$R_i(X) = LT(\hat{S}_i(X \oplus K_i)),\ i \in \{0, \ldots, 30\}$$
$$R_{31}(X) = \hat{S}_{31}(X \oplus K_{31}) \oplus K_{32}$$

and $\hat{S}_i$ is the application of the S-box $S_{(i\ (mod\ 8))}$ 32 times in parallel, and LT is the linear transformation.

The key scheduling algorithm of Serpent takes a 256-bit key as an input. If the key is shorter, then it is padded by a single bit of 1 and the remaining part is padded by bits of 0 up to 256 bits. By using an affine recurrence, the 256-bit key is used to construct 132 *prekeys* having length of 32 bits. The S-boxes are used to produce 32-bit keywords from prekeys. The round keys are obtained by combining these keywords.

4.2 Differential-Linear Cryptanalysis

In 1994, Langford and Hellman combined differential cryptanalysis with linear cryptanalysis and introduced differential-linear cryptanalysis [23]. They suggested using a truncated differential with probability 1 and concatenating a linear approximation with bias q (i.e. probability $1/2 + q$) where the output difference of the differential should contain zero differences in the places where input bits masked in the linear approximation. This way one can construct differential-linear distinguishers and the data complexity of the distinguisher is $O(q^{-4})$ chosen plaintexts. The exact number depends on the success probability and the number of possible subkeys.

Moreover, Biham, Dunkelman and Keller showed that it is possible to construct a differential-linear distinguisher where the differential holds with probability $p < 1$ and introduced enhanced differential-linear cryptanalysis [5]. They also showed that the attack is still applicable if the XOR of the masked bits of the differential is 1. In the enhanced method, the data complexity becomes $O(p^{-2}q^{-4})$ chosen plaintexts.

4.3 Differential-Linear Attacks on SERPENT

In [7] a differential-linear attack on 11-round Serpent-192 and Serpent-256 is presented. The attack combines the 3-round differential

$$\varDelta : 00000000000000000000000040050000 \rightarrow 0??00?000?000000000?00?0??0??0?0$$

that has a probability of $p = 2^{-7}$ with the 6-round linear approximation

$$\varLambda : 20060040000001001000000000000000 \rightarrow 00001000000000005000010000100001$$

of [3] that has bias $q = 2^{-27}$.

The first attack on 10-round Serpent-128 is also presented in [7] which is obtained by removing the last round of this linear approximation. The data and time complexities of these attacks are reduced in [17] by using the following improvements:

1. Better analysis of the bias of the differential-linear approximation,
2. Better analysis of the success probability,
3. Changing the output mask.

Moreover in [17], these reduced complexities are used to extend the 11-round attack and obtain the first 12-round attack on Serpent-256. In the following section we further improve these differential-linear attacks by using the differential factors of Serpent's S-boxes S_0 and S_1.

4.4 Improved Differential-Linear Attacks Using Differential Factors

The differential-linear attacks of [7,17] start at round 1 and the 3-round differential activates 5 S-boxes in this round. Two of the output differences of these activated S-boxes are 4_x and E_x which have differential factors as shown in Table 2. The authors guess every possible 20 subkey bits corresponding to these five S-boxes but the attacker can only obtain 18-bit advantage for this subkey due to Theorem 3 and there is no need to try half of the subkeys corresponding to these two S-boxes having differential factors. Thus, the advantage of the differential-linear attacks on 10, 11, and 12 rounds of Serpent are actually 38, 46, and 158 bits instead of 40, 48, and 160 bits, respectively. And again by Theorem 3, the same attacks can be performed with time complexities reduced by a factor of 4.

Moreover, the 12-round attack of [17] adds one more round to the top of the differential which affects every S-box at round 0 except the S-boxes 2, 3, 19, and 23 and guesses the 112 bits of the subkey corresponding to these active S-boxes. However, by using the undisturbed bits of Serpent, we observed that the output difference of the S-box 8 is exactly 4_x. Since $\mu = 4_x$ also has a differential factor for S_0, the attacker's advantage reduces to 157 bits and the time complexity of the attack further reduces by a factor of 2. Table 3 summarizes this 12-round attack and highlights the differential factors and the undisturbed bits that are used to reduce the time complexity.

We also observed that by replacing the 3-round differential with a more probable one, we can perform these attacks with less data complexity and capture four more subkey bits with a time complexity increased by a factor of $2^{3.35}$. These modified attacks are provided in Appendix B.

5 Conclusion

In this paper, we introduced a new S-box evaluation criteria that we call differential factors. Differential factors are mostly observed in small S-boxes like 3×3 and 4×4 which are preferred in hardware oriented lightweight block ciphers.

Table 3. 12-round differential-linear attack of [17]. Output differences μ that contain differential factors, which are 4_x and E_x for S_1 and 4_x for S_0, are shown in bold. Undisturbed bits are shown in italic.

Input	X_0:	????	????	0???	0???	????	????	????	00??
	X_1:	????	????	0???	0???	????	????	????	00??
	X_2:	????	????	0???	0???	????	???*1*	????	00??
	X_3:	????	????	0???	0???	????	????	????	00??
S_0	X_0:	??0?	00?0	0000	0?00	00?0	000**0**	00??	00??
	X_1:	??0?	????	00?0	0???	0???	???**0**	0?00	0000
	X_2:	000?	00??	0??0	0?00	??00	?00**1**	0?00	0000
	X_3:	?0??	?0??	00??	0???	??0?	0??**0**	?001	0000
LT	X_0:	?000	0000	0000	0??0	0?00	?000	0000	0000
	X_1:	?000	0000	0000	0??0	0?00	?000	0000	0000
	X_2:	?000	0000	0000	0??0	0?00	?000	0000	0000
	X_3:	?000	0000	0000	0*1*?0	0?00	*1*000	0000	0000
S_1	X_0:	0000	0000	0000	0100	0**0**00	**0**000	0000	0000
	X_1:	1000	0000	0000	0010	0**1**00	**0**000	0000	0000
	X_2:	0000	0000	0000	0000	0**1**00	**1**000	0000	0000
	X_3:	0000	0000	0000	0010	0**1**00	**0**000	0000	0000
LT	X_0:	0000	0000	0000	0000	0000	0000	0001	0000
	X_1:	0000	0000	0000	0000	0000	0000	0000	0000
	X_2:	0000	0000	0000	0000	0000	0000	1001	0000
	X_3:	0000	0000	0000	0000	0000	0000	0000	0000
9-Round Differential-Linear Characteristic $\Delta \circ \Lambda$									
Last Round									

We show that differential factors may reduce the attacked key space in differential cryptanalysis and its variants which results in an attack with reduced time complexity. As an example, we show that the differential factors of SERPENT's S-boxes are overlooked in Dunkelman *et al.*'s differential-linear attacks on SERPENT and the attacked round keys cannot be fully recovered in these attacks. We reduce the time complexities of these attacks by using the differential factors and provide the best differential-linear attacks on this cipher.

A Equivalent Definitions with only One Variable

When defining differential factors in Sect. 3, we used two variables x and y since they are directly linked to the input pairs in differential cryptanalysis. One can observe that the same definition and theorems of Sect. 3 for bijective S-boxes can be given by using a single variable. We provide them as follows.

Definition 7. *S has a* differential factor λ *for the output difference* μ *if*

$$S^{-1}(S(x) \oplus \mu) \oplus \lambda = S^{-1}(S(x \oplus \lambda) \oplus \mu)$$

for all x.

Proposition 1. *Definition 5 is equivalent to Definition 7.*

Proof. Since $S(x) \oplus S(y) = \mu$, we have $y = S^{-1}(S(x) \oplus \mu)$. Similarly, $y \oplus \lambda = S^{-1}(S(x \oplus \lambda) \oplus \mu)$ since $S(x \oplus \lambda) \oplus S(y \oplus \lambda) = \mu$. XORing both equations gives $\lambda = S^{-1}(S(x) \oplus \mu) \oplus S^{-1}(S(x \oplus \lambda) \oplus \mu)$ and we are done. □

Definition 8. *S has a* differential factor λ *for the output difference* μ *if*

$$S(S^{-1}(x) \oplus \lambda) \oplus \mu = S(S^{-1}(x \oplus \mu) \oplus \lambda)$$

for all x.

Proposition 2. *Definition 5 is equivalent to Definition 8.*

Proof. Let $y = S(x)$. Then the Definition 7 becomes

$$S^{-1}(y \oplus \mu) \oplus \lambda = S^{-1}(S(S^{-1}(y) \oplus \lambda) \oplus \mu)$$

for all y. Applying the S operation on both sides of the equation gives

$$S(S^{-1}(y \oplus \mu) \oplus \lambda) = S(S^{-1}(y) \oplus \lambda) \oplus \mu$$

for all y and we are done. □

Thus, Propositions 1 and 2 prove the Theorem 1.

Proposition 3. *If* λ_1 *and* λ_2 *are differential factors for an output difference* μ, *then* $\lambda_1 \oplus \lambda_2$ *is also differential factor for the output difference* μ*. i.e. All differential factors* λ_i *for* μ *forms a vector space.*

Proof. We have

$$S^{-1}(S(x) \oplus \mu) \oplus \lambda_1 = S^{-1}(S(x \oplus \lambda_1) \oplus \mu)$$

for all x, by Definition 7. And we have

$$S^{-1}(S(x \oplus \lambda_1) \oplus \mu) \oplus \lambda_2 = S^{-1}(S(x \oplus \lambda_1 + \lambda_2) \oplus \mu)$$

since λ_2 is a differential factor. Thus, we get

$$S^{-1}(S(x) \oplus \mu) \oplus \lambda_2 \oplus \lambda_2 = S^{-1}(S(x \oplus \lambda_1 \oplus \lambda_2) \oplus \mu)$$

for all x and we are done. □

B 3-Round Differentials with Higher Probability

The rounds of the 3-round differential used in the differential-linear attacks of [7,17] have probabilities 2^{-5}, 2^{-1}, and 1 but the authors observed experimentally that this differential has probability 2^{-7} instead of 2^{-6}. We observed that there are 3-round differentials of SERPENT with probability 2^{-5} that can be combined with the same linear approximations. The rounds of these differential have probabilities 2^{-5}, 1, and 1 and for this reason, the theoretical and practical probabilities of these differentials are the same. However, these differentials activate six S-boxes at the first round of the attack instead of five. So replacing the original differential with one of them results in capturing four more subkey bits but time complexity of the attacks also increases by a factor of 2^4.

Since the data complexity of a differential-linear attack is of $O(p^{-2}q^{-4})$ and replacing the differential result in $p = 2^{-5}$ instead of 2^{-7}, one would expect the modified attacks to have data and time complexities reduced by a factor of 2^4. However, experiment results show that the gain in the modified attacks is at most a factor of $(2^{-0.32})^2$. This is because the transition between the original differential and the linear approximation is far better than expected. For instance, when the original 3-round differential is combined with a 1-round linear approximation of bias 2^{-5}, Dunkelman *et al.* experimentally verified that the 4-round differential-linear path has bias $2^{-13.75}$, instead of $2 \cdot 2^{-7} \cdot (2^{-5})^2 = 2^{-16}$. We performed similar experiments on five different 3-round differentials with probability 2^{-5} using 2^{34} pairs and the results are summarized in Table 4.

Table 4. 4-Round biases for 3-round differentials with probability 2^{-5} and 1-round linear approximation with bias 2^{-5}.

#	Input Difference X_0 X_1 X_2 X_3 (in Hexadecimal)	#Active S-boxes	Bias	Standard Deviation
1	40000000 00000000 40000002 00000000	6	$2^{-13,49}$	$2^{-18.03}$
2	00000000 40000000 40000002 00000000	6	$2^{-13,43}$	$2^{-18.11}$
3	00000000 40000000 00000002 40000000	6	$2^{-13,56}$	$2^{-18.07}$
4	00000000 40000000 40000002 00000002	6	$2^{-13,43}$	$2^{-18.19}$
5	00000002 00000000 00000012 00000000	6	$2^{-14,65}$	$2^{-18.00}$

We replace the original differential with the second one from Table 4 and obtain new 10, and 11 round differential-linear attacks. This change provides a 4-round bias of $2^{-13,43}$ instead of $2^{-13.75}$. Thus the data and time complexity gain in the modified attack is a factor of $(2^{-0.32})^2$. This differential activates six S-boxes instead of five so we capture four more subkey bits and the time complexity is multiplied by 2^4. We summarize this modified attack in Table 5. Note that there are two differential factors for this differential, too. Since the rest of our modified attacks are almost identical to the attacks of [17], we refer the interested reader to [17].

Table 5. 11-Round differential-linear attack with a 3-round differential of probability 2^{-5}. Output differences $\mu = 4_x$ and $\mu = E_x$ that contain differential factors for S_1 are shown in bold. Undisturbed bits are shown in italic.

Step	Word	Difference	
Input	X_0:	0??0 0000 0000 00?0 0000 ?00? 00?0 0000	
	X_1:	0??0 0000 0000 00?0 0000 ?00? 00?0 0000	
	X_2:	0??0 0000 0000 00?0 0000 ?00? 00?0 0000	
	X_3:	0??0 0000 0000 00*1*0 0000 ?00? 00*1*0 0000	
S_1	X_0:	0000 0000 0000 0010 0000 000**0** 00**0**0 0000	
	X_1:	0110 0000 0000 0000 0000 100**1** 00**0**0 0000	
	X_2:	0000 0000 0000 0000 0000 000**1** 00**1**0 0000	
	X_3:	0000 0000 0000 0000 0000 100**1** 00**0**0 0000	
LT	X_0:	0000 0000 0000 0000 0000 0000 0000 0000	
	X_1:	0100 0000 0000 0000 0000 0000 0000 0000	↓
	X_2:	0100 0000 0000 0000 0000 0000 0000 0010	
	X_3:	0000 0000 0000 0000 0000 0000 0000 0000	$p = 2^{-5}$
S_2	X_0:	0000 0000 0000 0000 0000 0000 0000 0000	
	X_1:	0000 0000 0000 0000 0000 0000 0000 0010	
	X_2:	0100 0000 0000 0000 0000 0000 0000 0000	
	X_3:	0000 0000 0000 0000 0000 0000 0000 0010	
LT	X_0:	0000 0000 0000 0000 0000 0000 0000 0000	
	X_1:	0000 0000 0000 0000 0000 0000 0000 0000	
	X_2:	0000 0000 1000 0000 0000 0000 0000 0000	
	X_3:	0000 0000 0000 0000 0000 0000 0000 0000	
S_3	X_0:	0000 0000 ?000 0000 0000 0000 0000 0000	
	X_1:	0000 0000 ?000 0000 0000 0000 0000 0000	
	X_2:	0000 0000 ?000 0000 0000 0000 0000 0000	
	X_3:	0000 0000 ?000 0000 0000 0000 0000 0000	
LT	X_0:	00?0 0000 0000 ?000 0000 0??0 0?00 ?00?	
	X_1:	0000 ?00? 0000 0000 0000 0000 00?0 0000	
	X_2:	0000 0000 ?0?? 000? 0000 0000 000? 0?00	
	X_3:	0?00 0000 0000 0000 0?00 0000 0000 00?0	
S_4	X_0:	0??0 ?00? ?0?? ?00? 0?00 0??0 0??? ????	
	X_1:	0??0 ?00? ?0?? ?00? 0?00 0??0 0??? ????	
	X_2:	0??0 ?00? ?0?? ?00? 0?00 0??0 0??? ????	
	X_3:	0??0 ?00? ?0?? ?00? 0?00 0??0 0??? ????	
		6-Round Linear Approximation Λ	
		Last Round	

References

1. Biham, E., Anderson, R., Knudsen, L.R.: Serpent: a new block cipher proposal. In: Vaudenay, S. (ed.) FSE 1998. LNCS, vol. 1372, p. 222. Springer, Heidelberg (1998)
2. Biham, E., Biryukov, A., Shamir, A.: Cryptanalysis of Skipjack reduced to 31 rounds using impossible differentials. J. Cryptol. **18**(4), 291–311 (2005)
3. Biham, E., Dunkelman, O., Keller, N.: Linear cryptanalysis of reduced round serpent. In: Matsui, M. (ed.) FSE 2001. LNCS, vol. 2355, p. 16. Springer, Heidelberg (2002)
4. Biham, E., Dunkelman, O., Keller, N.: The rectangle attack - rectangling the serpent. In: Pfitzmann, B. (ed.) EUROCRYPT 2001. LNCS, vol. 2045, p. 340. Springer, Heidelberg (2001)

5. Biham, E., Dunkelman, O., Keller, N.: Enhancing differential-linear cryptanalysis. In: Zheng, Y. (ed.) ASIACRYPT 2002. LNCS, vol. 2501, pp. 254–266. Springer, Heidelberg (2002)
6. Biham, E., Dunkelman, O., Keller, N.: New results on boomerang and rectangle attacks. In: Daemen, J., Rijmen, V. (eds.) FSE 2002. LNCS, vol. 2365, p. 1. Springer, Heidelberg (2002)
7. Biham, E., Dunkelman, O., Keller, N.: Differential-linear cryptanalysis of serpent. In: Johansson, T. (ed.) FSE 2003. LNCS, vol. 2887, pp. 9–21. Springer, Heidelberg (2003)
8. Biham, E., Shamir, A.: Differential cryptanalysis of DES-like cryptosystems. J. Cryptol. **4**(1), 3–72 (1991)
9. Bilgin, B., Nikova, S., Nikov, V., Rijmen, V., Stütz, G.: Threshold implementations of all 3×3 and 4×4 S-boxes. In: Prouff, E., Schaumont, P. (eds.) CHES 2012. LNCS, vol. 7428, pp. 76–91. Springer, Heidelberg (2012)
10. Bogdanov, A., Knežević, M., Leander, G., Toz, D., Varıcı, K., Verbauwhede, I.: Spongent: a lightweight hash function. In: Preneel, B., Takagi, T. (eds.) CHES 2011. LNCS, vol. 6917, pp. 312–325. Springer, Heidelberg (2011)
11. 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)
12. Canniere, C.D., Sato, H., Watanabe, D.: Hash function Luffa: Specification. Submission to NIST (Round 2) (2009)
13. Chaum, D., Evertse, J.H.: Crytanalysis of DES with a reduced number of rounds: sequences of linear factors in block ciphers. In: Williams, H.C. (ed.) CRYPTO. LNCS, vol. 218, pp. 192–211. Springer, Heidelberg (1985)
14. 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)
15. Daemen, J., Peeters, M., Assche, G.V., Rijmen, V.: Nessie proposal: NOEKEON. NESSIE proposal, 27 October 2000
16. Dinur, I., Shamir, A.: Cube attacks on tweakable black box polynomials. In: Joux, A. (ed.) EUROCRYPT 2009. LNCS, vol. 5479, pp. 278–299. Springer, Heidelberg (2009)
17. Dunkelman, O., Indesteege, S., Keller, N.: A differential-linear attack on 12-round serpent. In: Chowdhury, D.R., Rijmen, V., Das, A. (eds.) INDOCRYPT 2008. LNCS, vol. 5365, pp. 308–321. Springer, Heidelberg (2008)
18. 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)
19. Helleseth, T. (ed.): Advances in Cryptology - EUROCRYPT 1993. LNCS, vol. 765. Springer, Heidelberg (1994)
20. Knudsen, L.R.: Truncated and higher order differentials. In: Preneel, B. (ed.) FSE. LNCS, vol. 1008, pp. 196–211. Springer, Heidelberg (1994)
21. Kocher, P.C., Jaffe, J., Jun, B.: Differential power analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, p. 388. Springer, Heidelberg (1999)
22. Kohno, T., Kelsey, J., Schneier, B.: Preliminary cryptanalysis of reduced-round Serpent. In: AES Candidate Conference, pp. 195–211 (2000)
23. Langford, S.K., Hellman, M.E.: Differential-linear cryptanalysis. In: Desmedt, Y.G. (ed.) CRYPTO 1994. LNCS, vol. 839, pp. 17–25. Springer, Heidelberg (1994)

24. Lim, C.H.: Crypton: A new 128-bit block cipher - specification and analysis (1998)
25. Lim, C.H.: A revised version of CRYPTON - CRYPTON V1.0. In: Knudsen, L.R. (ed.) FSE 1999. LNCS, vol. 1636, p. 31. Springer, Heidelberg (1999)
26. Matsui, M.: Linear cryptanalysis method for DES cipher. In: Helleseth, T. (ed.) EUROCRYPT 1993. LNCS, vol. 765, pp. 386–397. Springer, Heidelberg (1994)
27. McLaughlin, J., Clark, J.A.: Filtered nonlinear cryptanalysis of reduced-round serpent, and the wrong-key randomization hypothesis. In: Stam, M. (ed.) IMACC 2013. LNCS, vol. 8308, pp. 120–140. Springer, Heidelberg (2013)
28. National Bureau of Standards: Data Encryption Standard. FIPS PUB 46. National Bureau of Standards, U.S. Department of Commerce, Washington D.C., (15 January 1977)
29. Nguyen, P.H., Wu, H., Wang, H.: Improving the algorithm 2 in multidimensional linear cryptanalysis. In: Parampalli, U., Hawkes, P. (eds.) ACISP 2011. LNCS, vol. 6812, pp. 61–74. Springer, Heidelberg (2011)
30. Nyberg, K.: Differentially uniform mappings for cryptography. In: Helleseth, T. (ed.) EUROCRYPT 1993. LNCS, vol. 765, pp. 55–64. Springer, Heidelberg (1994)
31. Preneel, B., Takagi, T. (eds.): CHES 2011. LNCS, vol. 6917. Springer, Heidelberg (2011)
32. Saarinen, M.J.O.: Cryptographic analysis of all 4 × 4 s-boxes. In: Miri, A., Vaudenay, S. (eds.) Selected Areas in Cryptography. Lecture Notes in Computer Science, vol. 7118, pp. 118–133. Springer, Heidelberg (2011)
33. Schneier, B., Kelsey, J., Whiting, D., Wagner, D., Hall, C., Ferguson, N.: Twofish: A 128-bit block cipher. In: First Advanced Encryption Standard (AES) Conference (1998)
34. Selçuk, A.A.: On probability of success in linear and differential cryptanalysis. J. Cryptol. **21**(1), 131–147 (2008)
35. Shibutani, K., Isobe, T., Hiwatari, H., Mitsuda, A., Akishita, T., Shirai, T.: Piccolo: an ultra-lightweight blockcipher. In: Preneel, B., Takagi, T. (eds.) CHES 2011. LNCS, vol. 6917, pp. 342–357. Springer, Heidelberg (2011)
36. 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)
37. Tezcan, C.: Improbable differential attacks on PRESENT using undisturbed bits. J. Comput. Appl. Math. **259**, 503–511 (2014)
38. Tezcan, C., Taşkın, H.K., Demircioğlu, M.: Improbable differential attacks on SERPENT using undisturbed bits. In: Poet, R., Rajarajan, M. (eds.) Proceedings of the 7th International Conference on Security of Information and Networks, Glasgow, Scotland, UK, September 9-11, 2014. p. 145. ACM (2014)
39. V. Dolmatov (ed.): GOST 28147–89: Encryption, decryption, and message authentication code (MAC) algorithms. In: Internet Engineering Task Force RFC 5830 (March 2010)
40. Varici, K., Özen, O., Çelebi Kocair: Sarmal: Sha-3 proposal. Submission to NIST (2008)
41. 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)
42. Zhang, W., Bao, Z., Lin, D., Rijmen, V., Yang, B., Verbauwhede, I.: Rectangle: A bit-slice ultra-lightweight block cipher suitable for multiple platforms. IACR Cryptology ePrint Archive 2014, 84 (2014)
43. Zheng, Y. (ed.): Advances in Cryptology - ASIACRYPT 2002. Lecture Notes in Computer Science, vol. 2501. Springer, Heidelberg (2002)

Ciphertext-Only Fault Attacks on PRESENT

Fabrizio De Santis(✉), Oscar M. Guillen, Ermin Sakic, and Georg Sigl

Lehrstuhl für Sicherheit in der Informationstechnik,
Technische Universität München, Munich, Germany
{desantis,oscar.guillen,ermin.sakic,sigl}@tum.de

Abstract. In this work, we introduce fault attacks on PRESENT with faulty ciphertexts-only. In contrast to current differential fault attacks on PRESENT, which are mostly chosen-plaintext attacks, our fault attacks do not require the knowledge of the plaintexts to recover the secret key. This is a typical scenario when plaintexts are not easily accessible for the attacker, like in the case of smart devices for the upcoming Internet-of-Things (IoT) era where input data are mostly assembled within the cryptographic device, or when protocol-level countermeasures are deployed to prevent chosen-plaintext attacks explicitly. Our attacks work under the assumption that the attacker is able to bias the (nibble-wise) distribution of intermediate states in the final rounds of PRESENT by careful fault injections. To support our statements, we provide a detailed simulation analysis to estimate the practical attack complexities of (faulty) ciphertext-only fault attacks on PRESENT-80 discussing different fault injection scenarios. In the best case analysis (worst-case security scenario), only two faulty ciphertexts and negligible computational time are required to recover the entire secret key.

1 Introduction

Fault attacks are well-known implementation attacks introduced in the mid-nineties by Boneh et al. [12] to defeat the security of cryptographic protocols and devices. Since then, several fault attacks against symmetric [10,24,26] and asymmetric [9,12] cryptographic algorithms have been proposed in literature. Today, fault attacks still represent one of the major threats against the security of cryptographic implementations and the quest for new attacks and countermeasures is still a vivid area of research, as testified by the many conferences and recent developments [3,13]. The main idea behind fault attacks is to disrupt the normal functioning of a cryptographic device in order to induce errors during the cryptographic computations and exploit the associated (side-channel) information to finally recover the secret key. Nowadays, faults can be injected into cryptographic devices using several (low-budget) injection techniques such as: over-/under voltage feeding, injection of sudden transient changes (spikes) in the power supply lines [6,7], glitch insertion in the signal lines (*e.g.*, clock or reset line), high-energy injections using strong near-field Electro-Magnetic (EM) radiations [28] as well as light sources such as UV lamps, camera flashes [29] and lasers [30], or by simply bringing the device temperature outside the working

© Springer International Publishing Switzerland 2015
T. Eisenbarth and E. Öztürk (Eds.): LightSec 2014, LNCS 8898, pp. 85–108, 2015.
DOI: 10.1007/978-3-319-16363-5_6

ranges [18]. Like the more classical cryptanalytic attacks, also fault attacks can be classified according to how much knowledge about the input-output characteristic of a cryptosystem is given to the adversary: the vast majority of fault attacks known to date are either known-plaintext or chosen-plaintext attacks, while only very recently (faulty) ciphertext-only fault attacks were introduced in [14]. This class of attacks is the most generic one, as it only assumes that a collection of (faulty) ciphertexts obtained under the same secret key is known to the adversary.

As part of the new international standard for lightweight cryptography, named ISO/IEC 29192-2:2012 [19], PRESENT is one of the cryptographic algorithms of choice to secure Internet-of-Things (IoT) applications, specially for low-power and resource-constrained devices. IoT applications span from simple Radio-Frequency IDentification (RFID) tags, through Wireless Sensor Networks (WSNs) all the way to intelligent interfaces that connect and communicate within social, environmental, and user contexts [2,8]. While some RFID tags may use challenge-response authentication protocols, where an attacker could potentially have direct access to the input values of a cryptographic operation, in WSNs and more complex applications, the input values will come as a result of sensing and data processing, and therefore will typically not be easily accessible to the attacker (*e.g.*, due to privacy issues) [1]. Even though theoretically an attacker could gain enough knowledge of the internal operation of such cryptographic devices to try to infer the plaintexts, it is clear that such endeavor would require significant effort making it more efficient to run ciphertext-only attacks in practice.

In this work, we introduce fault attacks on PRESENT-80 and PRESENT-128 with faulty ciphertexts-only. In contrast to state-of-the-art fault attacks on PRESENT, our attacks do not require the knowledge of plaintext inputs to recover the secret key, and work under the assumption that the attacker is able to bias the (nibble-wise) distribution of intermediate states in the final rounds of PRESENT. These minimal attack assumptions are of particular relevance in the context of low-cost cryptographic devices, where simple protocol-level countermeasures based on input randomization [16] are typically deployed in place of more expensive algorithmic-level countermeasures based on redundant computations [16,22].

Previous Work. Differential fault attacks on PRESENT-80 were first introduced in [21]. They require about $2^{5.64}$ pairs of correct and faulty ciphertexts to retrieve the 64-bit last round key, plus an additional 2^{16} key search to recover the full 80-bit secret key. Subsequently, a differential fault attack on PRESENT-80's key-schedule algorithm was introduced in [32], exploiting a nibble-wise fault model. This attack requires 2^6 pairs of correct and faulty ciphertexts on average to recover 51-bit of the last round key and additional 2^{29} key search to recover the full 80-bit secret key. Improved differential fault attacks on PRESENT-80 and PRESENT-128 were described in [33], again exploiting a nibble-wise fault model. In the case of PRESENT-80, the attack requires 2^3 faulty pairs on average to reduce the key space to $2^{14.7}$, while in the case of PRESENT-128

the attack requires 2^4 faulty pairs on average to reduce the key space to $2^{21.1}$. State-of-the-art differential fault attacks on PRESENT-80 are provided in [4]. They require $2^{4.17}$ faulty ciphertexts on average and at least one pair of correct and faulty ciphertexts to recover the key with "negligible computational power". Finally, statistical differential fault attacks were introduced in [15]. The estimated overall attack complexity was $2^{36.3}$. All these fault attacks are either known-plaintext or chosen-plaintext and therefore they can not be mounted when the adversary has access only to the ciphertexts, like in most applications where data is assembled internally and transmitted to other devices encrypted (*e.g.*, IoT applications) or when input randomization countermeasures are deployed to explicitly chosen-plaintext attacks [16].

Our Contribution. In this work, we take the approach initiated by [14] and describe some fault attacks on PRESENT-80 and PRESENT-128 using faulty ciphertexts-only. Our attacks work under the weak and realistic assumption that the attacker is able to inject faults in at least one round and obtain nibble-wise faulty intermediate states in the final rounds of PRESENT[1]. Also, unlike the previous fault attacks on PRESENT, we do not require the adversary to be able to collect pairs of correct and faulty ciphertexts for the same plaintext and secret key, rather we assume that the plaintexts are unknown to the attacker and randomly-chosen for each cryptographic operation. We lay stress on the fact that all the fault attacks on PRESENT known so far [4,15,32,33] are not applicable in a (faulty) ciphertext-only scenario, as they all require the knowledge of at least one plaintext input.

Organization. We start introducing the notation and the necessary background information about PRESENT in Sect. 2. Then, we introduce faulty ciphertext-only attacks in Sect. 3. Our fault attacks on PRESENT are provided in Sect. 4, while a detailed simulation-based results analysis is provided in Sect. 5. Eventually, we conclude in Sect. 6.

2 Background and Notation

2.1 The PRESENT Block Cipher

PRESENT is a lightweight block cipher designed to fit ultra-constrained cryptographic devices [11]. The PRESENT block cipher is a SP-network and comes in two variants: PRESENT-80 with a block length of 64-bit and a key length of 80-bit, and PRESENT-128 with a block length of 64-bit and a key length of 128-bit. The datapath of both variants consists of 31 rounds and a final post-whitening key addition. Each round of the encryption algorithm consists of a bit-wise key addition (`addRoundKey`), a nibble-wise non-linear substitution layer

[1] The attacker can (possibly) exploit faulty intermediate states in different rounds obtained by injecting faults across different cryptographic computations and it is not required to inject multiple faults during the same cryptographic operation.

(`sBoxlayer`) and a bit-wise linear mixing layer (`pLayer`). The non-linear layer implements a 4×4-bit S-box operation which is applied 16 times to cover the full 64-bit state length. The linear mixing layer operates a bitwise permutation of the current state (*e.g.*, the bit at the jth position is moved to the $P(j)^{th}$ position according to the permutation table defined in [11, Sect. 3]). A sketch of the PRESENT-80 encryption algorithm is provided in Algorithm 1.

Algorithm 1. PRESENT-80 Encryption Algorithm

Input: p, K
Output: $c = \texttt{Enc}(p, K)$

1: $K_1, \ldots, K_{32} \leftarrow \texttt{KeySchedule80}(K)$ # generate round keys
2: $s \leftarrow p$
3: **for** $i = 1$ to 31 **do** # iterate the round function for 31 times
4: $\quad s \leftarrow \texttt{pLayer}(\texttt{sBoxlayer}(s \oplus K_i))$
5: **end for**
6: $c \leftarrow s \oplus K_{32}$ # final key whitening
7: **return** c

The two variants of the block cipher differ only in the key schedule algorithm (Line 1 of Algorithm 1). In the case of PRESENT-80, the 32 round keys $K_i = \kappa^i_{63}\kappa^i_{62}\ldots\kappa^i_0$ $(1 \leq i \leq 32)$ are obtained from the secret key $K = \kappa_{79}\kappa_{78}\ldots\kappa_0$ by applying the following transformations $(i-1)$ times and finally extracting the left-most 64-bit:

1. the key is cyclically shifted by 61 positions to the left
2. the left-most 4-bit of the key are passed through the 4×4-bit S-box
3. the round constant RC_i is bit-wise added to the key bits $\kappa_{19}\kappa_{18}\kappa_{17}\kappa_{16}\kappa_{15}$.

In the case of PRESENT-128, the 32 round keys $K_i = \kappa^i_{63}\kappa^i_{62}\ldots\kappa^i_0$ $(1 \leq i \leq 32)$ are derived from the secret key $K = \kappa_{127}\kappa_{126}\ldots\kappa_0$ by applying the following transformations $(i-1)$ times and finally extracting the left most 64-bit:

1. the key is cyclically shifted by 61 positions to the left
2. the left-most 8-bit of the key are passed through the implementation of two 4×4-bit S-boxes
3. the round constant RC_i is bit-wise added to the key bits $\kappa_{66}\kappa_{65}\kappa_{64}\kappa_{63}\kappa_{62}$.

Please refer to [11] for the full specification of the PRESENT block cipher.

2.2 Notation

Let $\widetilde{C}_r = \{\widetilde{c}^1_r, \widetilde{c}^2_r, \ldots, \widetilde{c}^n_r\}$ be the set of faulty ciphertexts obtained by injecting faults in the rth round with $1 \leq r \leq 31$. Let *ar*, *sb* and *pr* be the abbreviations for the `addRoundKey`, `sBoxlayer` and `pLayer` operations, respectively. Let $S^i_r(op)$ denote the intermediate state of the PRESENT block cipher right after the $op \in \{ar, sb, pr\}$ operation in the rth round during the ith execution. Let $\widetilde{S}^i_r(op)$ denote the faulty intermediate state and let $\widehat{S}^i_r(op, k)$ denote the hypothetical intermediate state obtained by applying a key guess k. For the sake of clarity,

the argument *op* will be omitted when referring generically to the intermediate state of any of the round operations. Let the suffix $[j]$ denote the jth *nibble* and the suffix $\{j\}$ denote the jth *bit* of the intermediate state (*e.g.*, $\texttt{pLayer}(\cdot)[j]$ denote the jth nibble after the linear mixing layer and $S_r^i\{j\}$ denote the jth bit of S_r^i). Finally, we denote by $1/Q_r$ the (nibble-wise) fault rate obtained in the rth round *e.g.*, $Q_r = 16$ if all the 16 nibbles of the state can be faulted simultaneously in the rth round or $Q_r = 1$ if only one nibble can be faulted in the rth round.

3 Faulty Ciphertext-Only Attacks

Faulty ciphertext-only attacks were introduced in [14] and mostly generalize the statistical differential fault attacks presented in [15,27]. The main difference between statistical differential fault attacks and fault attacks with faulty ciphertext-only is that the former exploits *differential* faulty distributions, for which the attacker is required to collect pairs of correct and faulty ciphertexts for the same plaintext and secret key values, while the latter exploits the distribution of faulty intermediate states only.

3.1 Statistical Distinguishers

Fault attacks with faulty ciphertext-only exploit the bias introduced in the distribution of intermediate states by careful fault injections. The secret key is recovered by evaluating the empirical distribution of hypothetical faulty intermediate states $\widehat{S}_r^i(k)$ for each key guess k by the means of a statistical distinguisher Δ. A statistical distinguisher Δ takes a set of faulty ciphertexts $\widetilde{C}$ as input and return a score value for each key guess indicating its eligibility as a candidate for the correct key. The key guess corresponding to the highest or the lowest score is typically chosen as candidate for the correct key. In practice, there exists several possible statistical distinguishers and one would aim at using the most efficient one, that is, the distinguisher which is able to identify the correct key with the lowest number of faulty ciphertexts.

The choice of the distinguisher typically depends on how much previous knowledge the adversary has about the distribution of faulty intermediate states. If the attacker knows the distribution $p_{k^*}(\cdot)$ of the faulty intermediate states $\widetilde{S}_r$ (or, more generally, the distribution $f_{k^*}(\cdot)$ of any proper function $\zeta(\cdot)$ of the faulty intermediate state $\widetilde{S}_r$ *e.g.*, the Hamming weight), then the attacker can efficiently identify the correct key using a maximum likelihood based distinguisher Δ_{ML} [27], by evaluating the likelihood function of the hypothetical intermediate states for each key guess k:

$$\arg\max_k \prod_{i=0}^{n-1} f_{k^*}\left(\zeta(\widehat{S}_r^i(k)[j])\right)$$

Most frequently, it is not possible for the attacker to profile the distribution of the faulty intermediate states in advance. In this case, the attacker can try

to identify the correct key k^* under the assumption that the distribution of the faulty intermediate states $f_k(\cdot)$ for all the wrong-key guesses $k \neq k^*$ is close to a supposed distribution $q(\cdot)$, while the distribution of the faulty intermediate state for the correct-key guess $f_{k^*}(\cdot)$ is far from $q(\cdot)$. This assumption is generally referred in literature as the *wrong-key assumption* [17,20,27]. In this case, the attacker can try to distinguish the correct key using a χ^2-test based distinguisher Δ_{χ^2} by evaluating the test for each key guess k:

$$\arg\max_k \sum_{x \in \Im(\zeta)} q(x) \left(f_k(x) - q(x)\right)^2,$$

where $\Im(\zeta)$ denotes the common sample space defined by the image of ζ. In case $q(\cdot)$ is chosen to be the discrete uniform distribution, then the χ^2 test-based distinguisher is also called Square Euclidean Imbalance (SEI) in cryptographic literature [14,20,27].

Based on the same principle, yet another option consists in evaluating just a characteristic of the faulty intermediate distributions under the assumption that a distinguishable characteristic does exist for the correct key only. For instance, it is possible to try distinguishing the correct key by evaluating the sample mean for each key guess k, as follows:

$$\arg\min_k \frac{1}{n} \sum_{i=0}^{n-1} \zeta(\widehat{S}_r^i(k)[j])$$

When $\zeta(x)$ is chosen to be the Hamming weight of x, this latest distinguisher corresponds to the minimum average Hamming weight distinguisher Δ_{HW} used in [14]. In practice, many other distinguishers could be used, but hereinafter we will consider Δ_{ML}, Δ_{χ^2} and Δ_{HW} only.

3.2 Fault Models

Ciphertext-only fault attacks require biasing the distribution of intermediate states away from their original distribution determined by the specific cryptographic algorithm. We model the fault injection repeatability of a given injection setup with the injection probability $1/\eta$. Then, we summarize the attacker's fault injection capabilities in the following three fault models:

Fault Model 1. "$\{0,1\} \rightarrow 0$ bit-clears" with probability $1/\eta$:

$$\Pr\left(\widetilde{S}_r^i[j] = S_r^i[j] \wedge c\right) = \frac{1}{\eta},$$

where $c \in \mathbb{Z}_{16} \backslash \{15\}$ is a constant value that specifies which bits get cleared and the $\wedge$ represents the boolean AND operation (*e.g.*, $c = (0011)_2$ clears the first two most significant bits of $S_r^i[j]$, while leaving the two least significant bits untouched).

Fault Model 2. "1 → 0 bit-flips" with probability $1/\eta$:

$$\Pr\left(\widetilde{S}_r^i[j] = S_r^i[j] \wedge (m_r \vee c)\right) = \frac{1}{\eta},$$

where $m_r \xleftarrow{\$} \{0,1\}^4$ is a uniform randomly drawn mask, the $\vee$ operator defines the boolean OR operation and $c \in \mathbb{Z}_{16}\backslash\{15\}$ is a constant value that specifies which bits *might* get flipped (*e.g.*, for $c = (0011)_2$ the first two most significant bits of $S_r^i[j]$ might get flipped depending on the first two most significant random bits of m_r. The value of c is a fixed property for a given injection setup, while the value of m_r can change at every injection.

Fault Model 3. Combined "$\{0,1\} \to 0$ bit-clears" and "1 → 0 bit-flips" with probability $1/\eta$:

$$\Pr\left(\widetilde{S}_r^i[j] = (S_r^i[j] \wedge m_r \wedge c)\right) = \frac{1}{\eta},$$

where $m_r \xleftarrow{\$} \{0,1\}^4$ is a uniform randomly drawn mask and $c \in \mathbb{Z}_{16}\backslash\{0,15\}$ is a constant value that specifies which bits get cleared and which bits might get flipped as in the previous fault models.

All the three fault models allow for complementary fault models (*e.g.*, "$\{0,1\} \to 1$ bit-sets" or "0 → 1 bit-flips") which can be obtained simply by swapping the operations $\wedge$ and $\vee$ and by taking *e.g.*, $c \in \mathbb{Z}_{16}\backslash\{0\}$ from the original models. We argue that these models have been proven practical several times in practice using (low-budget) fault injection equipments: *Fault Model 1.* has been proved experimentally feasible in [30], while *Fault Model 2.* and *Fault Model 3.* have been shown practical in [5] by careful glitch injections into the clock line. Similarly, the complementary "$\{0,1\} \to 1$ bit-sets" or "0 → 1 bit-flips" fault models have been obtained in practice *e.g.*, in [23] using high-energy near-field radiations or in [5,25] by inserting glitches into pre-charged buses. Finally, please note that our model possibly allows for different fault models on different nibbles, hence allowing for *e.g.*, bit-flips at different positions for different nibbles.

4 Faulty Ciphertext-Only Attacks on PRESENT

Fault attacks typically exploit faulty computations to recover the last round key first and then invert the key schedule algorithm to finally recover the secret key. However, in the case of PRESENT, recovering the last round key is not sufficient to invert the key schedule algorithm and recover the secret key. More precisely, in order to invert the key schedule algorithm of PRESENT from a single round key, an additional 2^{15} average guess work would be required in the case of PRESENT-80, while an additional 2^{63} average guess work would be required in the case of PRESENT-128. However, in a (faulty) ciphertext-only scenario, guessing the

correct key *e.g.*, for the case of PRESENT-80, is not straightforward, as the attacker does not have access to the plaintexts to quickly verify the key guesses. Yet, it *could* be possible to verify different key guesses under the assumption that the distribution of plaintexts for the correct key guess is somewhat different than the one for a wrong key guess. In case this assumption is valid, then the attacker could *e.g.*, estimate the distribution of plaintexts for each key guess and select the key guess which leads to the set of plaintexts with the smallest entropy - that is, under the assumption that wrong-key guesses maximize the entropy of plaintexts, while the correct key minimizes it. Although appealing, this approach would require a very large number of ciphertexts depending on intrinsic difficulty of estimating the entropy of plaintexts in the particular application case (*e.g.*, depending on whether randomization countermeasures like in [16] are deployed or not).

A more suitable and convenient approach for the attacker is to recover multiple (consecutive) round keys in the last rounds of PRESENT and invert the key schedule algorithm using multiple round keys. In the case of PRESENT-80, the key schedule can be inverted using *two* consecutive 64-bit round keys, say the post-whitening key K_{32} and the last round key K_{31}, while in the case of PRESENT-128, the key schedule can be inverted using *three* consecutive 64-bit round keys, say the post-whitening key K_{32}, the last round key K_{31} and the second to last round key K_{30}.

In the following subsections, we develop two attack strategies which mainly differ in whether they exploit faulty nibbles located across multiple rounds or just in a single round. The first strategy recovers the round keys one after another by iteratively exploiting faulty nibbles starting from the last round and peeling off one round after another. Please note that this strategy only requires faulty states across different rounds, but does not require to inject multiple faults during the same cryptographic computation. The second strategy is based on exploiting faulty nibbles from one single round only and by recovering multiple consecutive round keys at the same time. For each attack, we compute the time complexity τ and data complexity δ requirements as a function of the number of faulty ciphertexts $N_{\Delta}^{\mathcal{A},\mathcal{S}}$ required to distinguish the distribution of a faulty nibble for the correct key. We will estimate actual values of $N_{\Delta}^{\mathcal{A},\mathcal{S}}$ by simulations in Sect. 5. To lighten the notation, we will simply refer to N (instead of $N_{\Delta}^{\mathcal{A},\mathcal{S}}$) in the next subsections. However, please note that the number of faulty ciphertexts required to distinguish the correct key depends on the specific considered attack $\mathcal{A}$, the fault injection setup $\mathcal{S}$ and the intrinsic efficiency of the chosen statistical distinguisher Δ. For the sake of compactness, we assume that each target nibble requires the same number of faulty ciphertexts for a given triple $< \mathcal{A}, \mathcal{S}, \Delta >$ in our analysis. In practice, the veracity of this assumption depends on the specific fault injection setup $\mathcal{S}$ which specifies in which the round the attacker is able to inject faults, the fault model, the fault rate $1/Q_r$ and the fault injection probability $1/\eta$.

4.1 Fault Attacks on PRESENT-80

Fault Attacks Across Multiple Rounds. The secret key can be recovered by exploiting faulty intermediate states located across multiple rounds by peeling off one round after the other. This strategy exploits faulty nibbles located after either one of the `addRoundKey`, `sBoxlayer` or `pLayer` operations in both the 31st round and in the 30th round. It requires two sets of faulty ciphertexts $\widetilde{C}_{30}$ and $\widetilde{C}_{31}$ which can be (possibly) obtained faulting the cryptographic computations in different rounds.

Assuming that the state after the `sBoxlayer` is targeted, then we can express every nibble $j \in [0, 15]$ of the hypothetical intermediate state $\widehat{S}^i_{31}(sb)[j]$ as a function of the faulty ciphertexts $\widetilde{c}^i_{31}$ and post-whitening key guess $\widehat{K}_{32}$ as follows:

$$\begin{aligned} \widehat{S}^i_{31}(sb)[j] &= \texttt{pLayer}^{-1}(\widetilde{c}^i_{31} \oplus \widehat{K}_{32})[j] \\ &= \texttt{pLayer}^{-1}(\widetilde{c}^i_{31})[j] \oplus \widehat{k}_{32}[j] \end{aligned}, \tag{1}$$

where $\widehat{k}_r[j] = \texttt{pLayer}^{-1}(\widehat{K}_r)[j]$ denote the jth-nibble of the round key guess $\widehat{K}_r$ when the inverse `pLayer` operation is applied to it. This simple equation allows to recover the 16 nibbles of the post-whitening key K_{32} by 16 independent search using 2^4 key nibble guesses, leading to a time complexity of $2^4 * 16 * N = 2^8 * N$ and a data complexity $\lceil 16/Q_{31} \rceil * N$ to recover the post-whitening key K_{32} from faulty intermediate states in the 31st round. The next step is to repeat the attack by exploiting the (nibble-wise) faulty states in the 30th round to recover the 16 nibbles of the last round key K_{31} and finally invert the key schedule algorithm. Thus, by assuming that the post-whitening key K_{32} has been successfully recovered using Eq. (1), the intermediate state $S^i_{31}(ak)$ can be computed backwards from the faulty ciphertexts $\widetilde{c}^i_{30}$. Then, each nibble of the last round key K_{31} can be recovered by 16 independent searches $\forall j \in [0, 15]$ as before:

$$\widehat{S}^i_{30}(sb)[j] = \texttt{pLayer}^{-1}(S^i_{31}(ak))[j] \oplus \widehat{k}_{31}[j] \tag{2}$$

Assuming that the number of faulty ciphertexts N required to successfully attack the nibbles in the 30th round is the same as the attack in the 31st round as well as their fault rates $Q = Q_{31} = Q_{30}$, then the time and data complexity requirements of the attack are simply doubled, being $2^9 * N$ and $2 * \lceil 16/Q \rceil * N$, respectively. This attack strategy can be improved if the faulty nibbles of the 30th round are located after the `pLayer` operation. In this case, it is possible to target directly the 5 least significant nibbles of K_{31} only, which are sufficient to invert the key schedule together with K_{32}, leading to an improved data complexity $\left(\lceil 5/Q_{30} \rceil + \lceil 16/Q_{31} \rceil\right) * N$ and time complexity of $(2^8 + 5 * 2^4) * N \approx 2^{8.39} * N$. The total time complexity can be further reduced down to $\approx 2^{8.29} * N$ by observing that 4 bit of the 5 faulted nibbles of K_{31} are actually known from K_{32}.

Fault Attacks in One Single Round Only. Yet another strategy is to recover the secret key from faulty intermediate state located in one single round only. In this case, one very first option is to proceed as in Eq. (1) by recovering the

post-whitening key K_{32} and verifying the entropy of plaintexts for the remaining 2^{16} key guesses instead of proceeding backwards. This attack has a time complexity of $N * 2^8 + C * 2^{16}$ and a data complexity of $N * Q_{31} + C$, where C is minimum number of *correct* ciphertexts required to properly estimate the entropy of plaintexts for each key guess.

As discussed previously, a more suitable approach in the context of faulty-ciphertext only attacks is to recover the 32nd and 31st round keys simultaneously. In this case, there are at least two possible options depending on whether the faulty nibbles are located either after the `addRoundKey`/`sBoxlayer` operations or after the `pLayer` operation in the 30th round.

The first attack option requires faulty nibbles after the `addRoundKey` or after the `sBoxlayer` operation in the 30th round. In this case, all the 16 nibbles of the post-whitening key K_{32} and 16 bits of the last round key K_{31} can be recovered simultaneously with a time complexity of $N*4*2^{20} = N*2^{22}$ by four independent searches on 20 key bits each. Assuming that the faulty nibbles are located after the `sBoxlayer` operation, we can describe the four independent searches $\forall j \in \{0, 1, 2, 3\}$ and $q \in \{0, 4, 8, 12\}$ by the following system of equations:

$$\begin{cases} T^i[j+q] & = \texttt{pLayer}^{-1}(\tilde{c}^i_{30})[j+q] \oplus \widehat{k}_{32}[j+q] \\ U^i[j+q] & = \texttt{sBoxlayer}^{-1}(T^i)[j+q] \\ \widehat{S}^i_{30}(sb)[4*j] & = \texttt{pLayer}^{-1}(U^i)[4*j] \oplus \widehat{k}_{31}[4*j] \end{cases} \tag{3}$$

In order to recover the remaining $12 * 4 = 48$ key bits of the last round key K_{31}, Eq. (2) can be used as the post-whitening key K_{32} has been fully recovered after Eq. (3). This leads to a time complexity of $N * 4 * (2^{20} + 3 * 2^4) \approx N * 2^{22}$.

The second attack option requires faulty nibbles after the `pLayer` operation in the 30th round. In this case, the nibbles of the round keys K_{32} and K_{31} can be recovered two at a time simultaneously leading to a reduced time complexity of $N * (2^8 * 16) = N * 2^{12}$. In fact, in this case, it is possible to write the following simplified equations $\forall j \in [0, 15]$:

$$\widehat{S}^i_{30}(pr)[j] = \texttt{sBoxlayer}^{-1}(\texttt{pLayer}^{-1}(\tilde{c}^i_{30} \oplus \widehat{K}_{32}))[j] \oplus \widehat{K}_{31}[j]$$

In all the cases the data complexity is $\lceil 16/Q_{30} \rceil * N$. Please refer to Figs. 4 and 5 in Appendix A for a visualization of the fault propagation pattern and the corresponding necessary guess work for these attacks.

4.2 Fault Attacks on PRESENT-128

The attacks on PRESENT-128 mostly resemble and extend the attack strategies previously presented for PRESENT-80, as the only difference between the two variants of the PRESENT block cipher lies in the key-schedule algorithm.

Fault Attacks in Multiple Rounds. The first attack strategy consists in exploiting any of the previously presented multiple rounds attacks on PRESENT-80 to recover the post-whitening key K_{32} and the last round key K_{31} and then

testing the entropy of plaintexts for a reduced set of 2^3 key guesses, only. In the best considered case (worst-case security), this attack has a time complexity of $N * 2^{8.29} + 2^3 * C$ and a data complexity of $\left(\lceil 5/Q_{30}\rceil + \lceil 16/Q_{31}\rceil\right) * N + C$.

Alternatively, a second attack strategy it to exploit one more faulty nibble after either the `addRoundKey` or after the `sBoxlayer` operation in the 29th round in order to recover the least significant nibble of the second-to-last round key K_{30}, once the the post-whitening key K_{32} and the last round key K_{31} have been successfully recovered. In the best considered case (worst-case security), this attack has a time complexity of $N*(2^{8.29}+2^4) \approx N*2^{8.36}$ and a data complexity of $\left(\lceil 5/Q_{30}\rceil + \lceil 16/Q_{31}\rceil\right) * N + N$.

Fault Attacks in a Single Round Only. As in the previous case, one first strategy would consist in exploiting any of the previously presented single rounds attacks on PRESENT-80 to recover the post-whitening key K_{32} and the last round key K_{31} simultaneously and then testing the entropy of plaintexts for a reduced set of 2^3 key guesses, only. In the best considered case (worst-case security), this attack has a time complexity of $N * 2^{12} + 2^3 * C$ and a data complexity of $\lceil 16/Q_{30}\rceil * N + C$.

As a second alternative strategy, it is possible to exploit the faulty nibbles located after the `pLayer` operation in the 29th round by four independent to recover the post-whitening key K_{32}, the last round key K_{31} and the second to last round key K_{30} simultaneously. This strategy has a time complexity of $4 * N * (2^{24} + 3 * 2^8) \approx N * 2^{26}$ and a data complexity of $\lceil 16/Q_{29}\rceil * N$. The attack can be visualized with the help of Fig. 6 in Appendix A.

5 Simulation Analysis

Based upon the distinguishers, the fault models and the attacks, presented in the previous sections, we performed several simulations to estimate the number of faulty ciphertexts $N_{\Delta}^{\mathcal{A},\mathcal{S}}$ required to a successful key-recovery fault attack. As a metric to estimate $N_{\Delta}^{\mathcal{A},\mathcal{S}}$, we computed the guessing entropy and the success rate [31] over 100 trials, and defined $N_{\Delta}^{\mathcal{A},\mathcal{S}}$ as the number of faulty ciphertexts required to obtain a 100 % success rate or, equivalently, a zero guessing entropy. In the following, we discuss the results of our simulations for different configurations of the triple $< \mathcal{A}, \mathcal{S}, \Delta >$ in the case of PRESENT-80 only. We expect similar considerations to apply for the case of PRESENT-128 as well.

Fault Models. Faulty ciphertext-only fault attacks require faulty intermediate states in the final rounds of PRESENT in order to recover the last round keys. This means that faults can be injected either directly in the final rounds or even in previous middle rounds and let them propagate until the final rounds. In the first case, under the reasonable assumption that the intermediate values are uniformly distributed in the final rounds of PRESENT, the distributions of faulty intermediate values depend only on the fault model and the injection probability (which only make the distributions more "noisy"). The small size

of our nibble-wise fault models actually allows for the enumeration of all the possible fault distributions for the three considered fault models described in Sect. 3.2, which are provided in Appendix B (Figs. 7, 8, and 9) for the convenience of the reader. In the second case, instead, the distributions depend not only on the previous parameters, but also on the specific look of intermediate cryptographic operations, the fault rate (number of nibbles which can be faulted per cryptographic operation) as well as through how many rounds the faults are propagated. Considering the worst-case security scenario, where the attacker is able to inject faults directly in the final rounds with probability 1 using the *Fault Model 1.* with $c = 0$ and $Q = 16$ (the intermediate states are cleared entirely), faulty ciphertext-only attacks over multiple rounds have a data complexity of 2 (resp. time complexity of $2^{9.29}$) using either Δ_{ML} or Δ_{HW}, which significantly improve state-of-the-art fault attack complexities (cf. Sect. 2).

Considering the more relaxed *Fault Model 2.*, instead, Fig. 1 shows the guessing entropy for all the possible values of c obtained using the average Hamming weight Δ_{HW} as a distinguisher. Interestingly, the success probability of the attacks in this case does not only depend on how many bits are flipped, but also on which particular bits are flipped. Similar results and considerations apply for *Fault Model 3.* being just a combination of bit clears and bit flips.

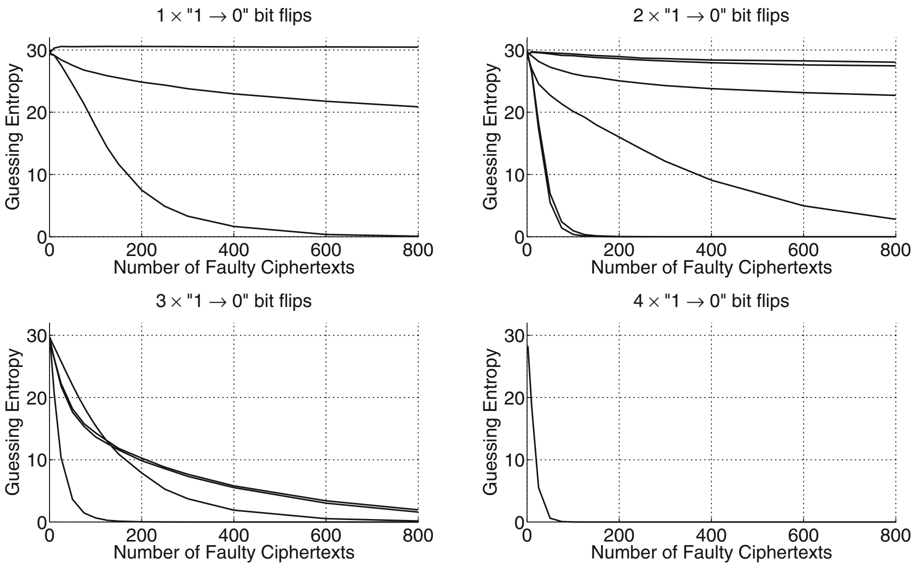

Fig. 1. Guessing entropy for attacks on the last two rounds of PRESENT using *Fault Model 2.* for all the possible values of c, Δ_{HW}, injection probability 1, fault rate $Q = 16$.

Single vs. Multiple-Round Attacks. Single round attacks have the advantage to require faulty intermediate states in only a single round, but on the other side

they have an increased time complexity as described in Sect. 4. In Fig. 2, we report the success rate of attacks computed over single vs. multiple rounds in the case of *Fault Model 2.* with $c = 0$ and $Q = 16$ (the intermediate states are cleared entirely) using the average Hamming weight distinguisher Δ_{HW}. The confidence intervals are computed using the normal approximation method for a confidence level of 95 % and are depicted using dashed gray lines. It can be observed that the two attacks converge pretty fast to the asymptotic value, while single round attacks have a larger variance, multiple round attacks requires a smaller number of faulty ciphertext to distinguish the correct key. However, please note the multiple round attacks have always doubled data complexity requirements by construction, as they have to iterate the attack at least twice to recover the last two round keys. Hence, the attacks have almost the same data complexity requirements.

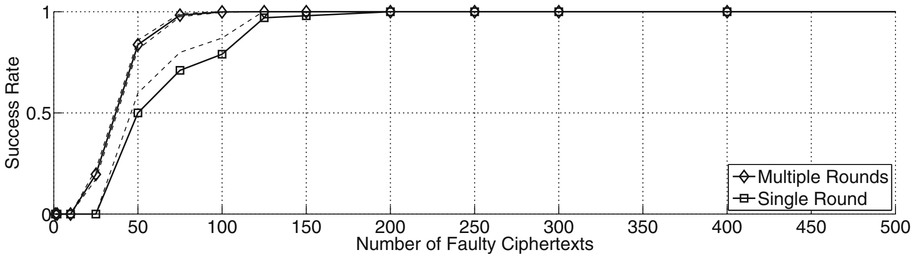

Fig. 2. Success rate and guessing entropy for attacks over single vs. multiple rounds using Δ_{HW}, *Fault Model 2.*, $c = 0$, injection probability 1, fault injection rate $Q = 16$.

Distinguishers. The maximum likelihood based distinguisher Δ_{ML} represents the worst-case scenario analysis, where it is assumed that the attacker is able to profile the distribution of faulty intermediate states determined by the given fault injection setup (*e.g.*, from an identical copy of the target of evaluation or from running simulations). However, depending on the particular fault model, other distinguishers might have similar performance. For instance, in the case of *Fault Model 2.* with $c = 0$, the distribution of faulty intermediate state which are clearly biased towards zero, hence leading the average Hamming weight distinguisher Δ_{HW} to perform comparably to Δ_{ML}. On the other side, decreasing the fault injection probability, Δ_{ML} performs much better in distinguishing the correct key as the average Hamming weight distinguisher would not be able to consider the noise introduced by the mixture of correct uniformly distributed intermediate values and faulty biased intermediate values. Finally, the Δ_{χ^2} distinguisher has been revealed to be totally ineffective in many of the considered cases, as it practically happens that the distributions of faulty intermediate states, though different for different key guesses, they all lay at the same “distance” from the uniform distribution. This effect has been observed also in [14]

and it is actually caused by the bijective property of the cryptographic operations involved in the backward computations of faulty intermediate values. In this case, backward computations only lead to permutations of the values under different key guesses (the score value computed by Δ_{χ^2} is the same value for all the key guesses), which breaks the wrong-key assumption and therefore make the correct key indistinguishable from the other key guesses. Hence, in order to be able to use Δ_{χ^2} to distinguish the correct key effectively, an additional requirement on the distributions of faulty intermediate values must be required, that is, the distribution of faulty intermediate values must be different from a given reference distribution (*e.g.*, the uniform distribution) *and* not symmetric under different key guesses.

Injection Probability. Finally, we evaluate the effect of different injection probabilities $p \in \{1, 0.5, 0.33, 0.25\}$ on the success rate of multi-round attacks using the average Hamming weight distinguisher as depicted in Fig. 3. We observe that, depending on the considered fault model, the data complexity scales differently with the injection probability. This effect can be explained by looking at the distribution of faulty intermediate values which, depending on the fault model, are more or less biased towards zero.

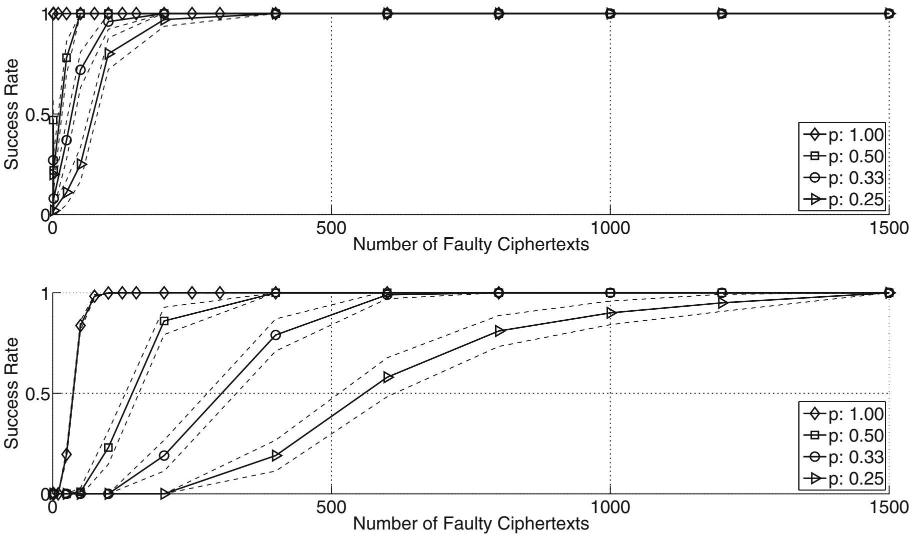

Fig. 3. Success rate for different injection probabilities $p \in \{1, 0.5, 0.33, 0.25\}$ using Δ_{HW}, *Fault Model 1.*, $c = 0$ (top), *Fault Model 2.*, $c = 0$ (below), multiple rounds attacks, injection rate $Q = 16$.

6 Conclusion

In this work, we have introduced fault attacks on PRESENT-80 and PRESENT-128 with (faulty) ciphertexts-only and provided a detailed simulation analysis discussing different injection setups. In contrast to state-of-the-art differential fault attacks on PRESENT, faulty ciphertext-only attacks do not require the knowledge of plaintext values. These minimal attack requirements make ciphertext-only fault attacks of particular interest when considering the security of low-cost cryptographic devices, as they can completely defeat the security of typical low-cost fault attack countermeasures such as input randomization-based countermeasures. Indeed, we have shown by simulations that faulty ciphertext-only attacks on PRESENT-80 have considerably low time and data complexities in many attack scenarios and they are even more efficient than state-of-the-art fault attacks when worst-case security analysis is considered (average data complexity is 2 and time complexity is $2^{9.29}$). In order to assess the real threat in a practical scenario, future work will comprise performing various fault injection attacks on low-cost cryptographic devices to compare experimental and simulations results.

Finally, we lay stress on the fact that faulty ciphertext-only attacks are not limited to the fault models and distinguishers presented in this work. In principle, any fault model which can lead to a bias of the intermediate states of PRESENT in the final rounds and any distinguisher which can effectively discern the bias can be used. This fact leaves an interesting research question open for further research, that is: up to which round faults can be injected and which fault models remain valid in order to obtain *exploitable* faulty intermediate states in the final rounds of PRESENT. This question has no trivial answer as the distributions obtained in the final rounds of PRESENT from fault injections in middle rounds will depend on many parameters such as the number of rounds, the fault model, the specific look of cryptographic operations and the fault injection rate. Most importantly, fault injections in the middle rounds should not only deliver distributions which are biased in the final rounds of PRESENT, but also which can be exploited by the considered distinguishers *e.g.*, they must be independent of the secret key and not symmetric under different key hypotheses.

Acknowledgements. The authors would like to thank the anonymous reviewers for their valuable comments and suggestions. This work has been funded in part by the German Federal Ministry of Education and Research 163Y1200D (HIVE).

A Fault Propagation in the Datapath of PRESENT

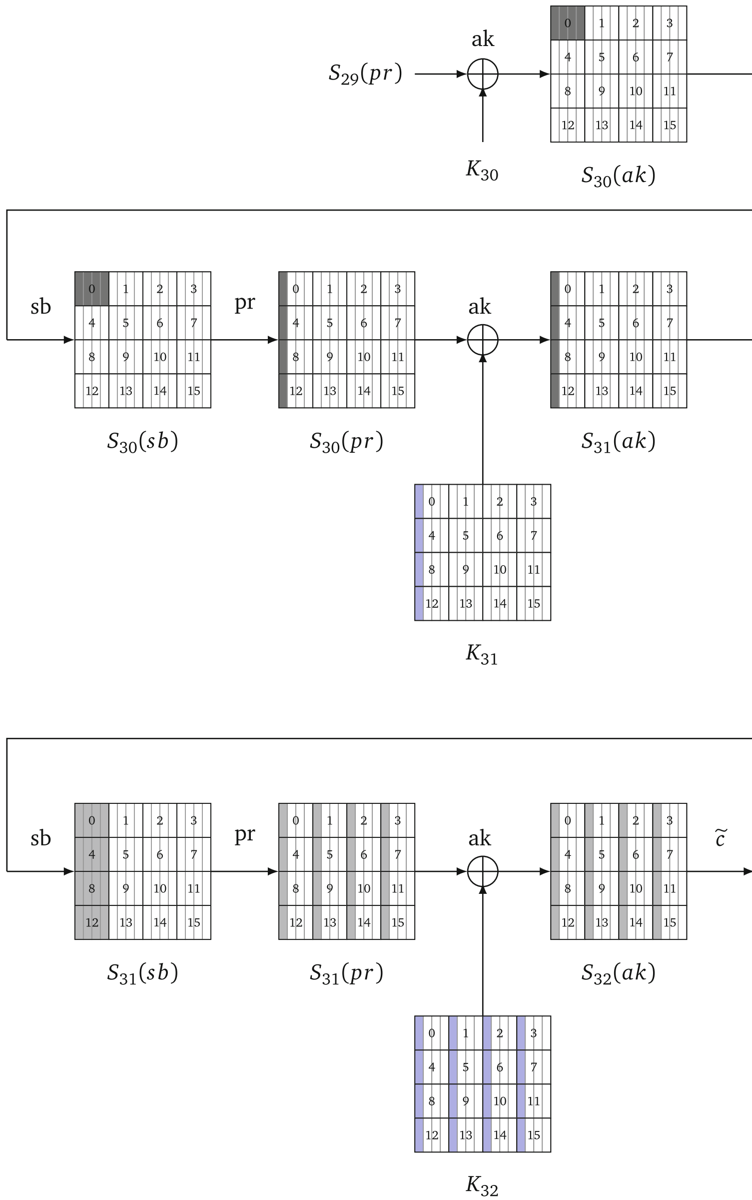

Fig. 4. Fault propagation with fault injection after the `addRoundKey` operation in the 30th round. Gray: faulted bits. Blue: key bits to guess (Color figure online).

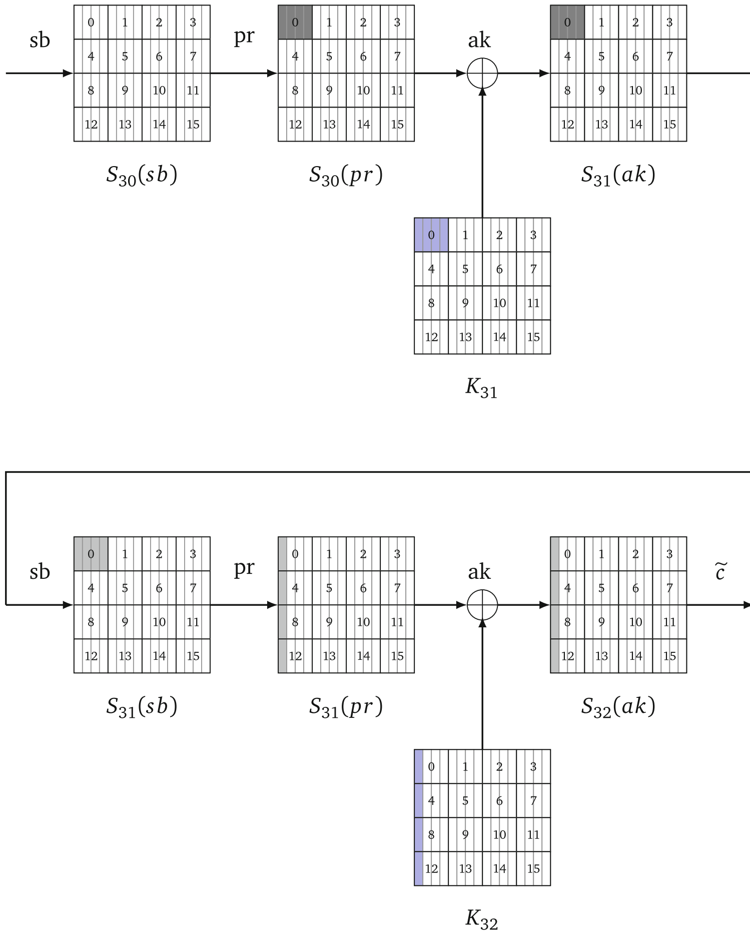

Fig. 5. Fault propagation with fault injection after the `pLayer` operation in the 30th round. Gray: faulted bits. Blue: key bits to guess (Color figure online).

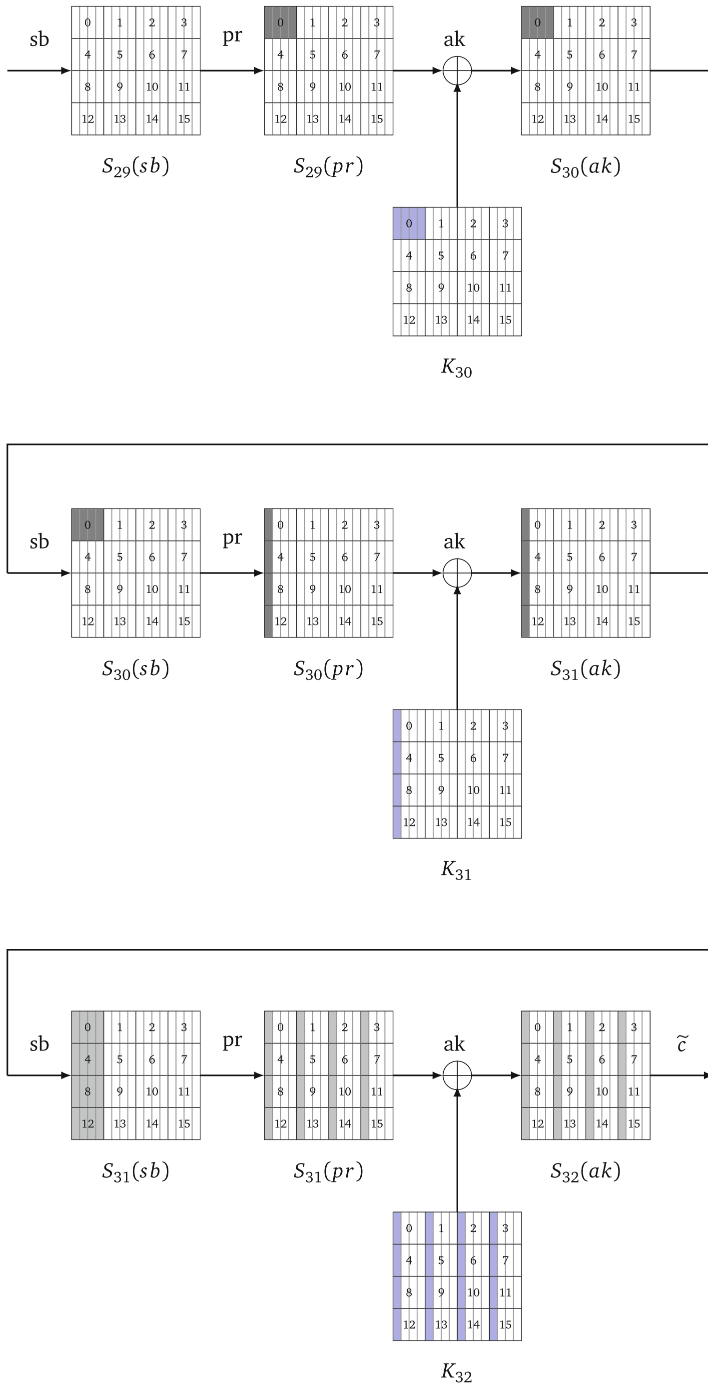

Fig. 6. Fault propagation with fault injection after the `pLayer` operation in the 29th round. Gray: faulted bits. Blue: key bits to guess (Color figure online).

B Probability Distributions of Faulty Intermediate Nibbles

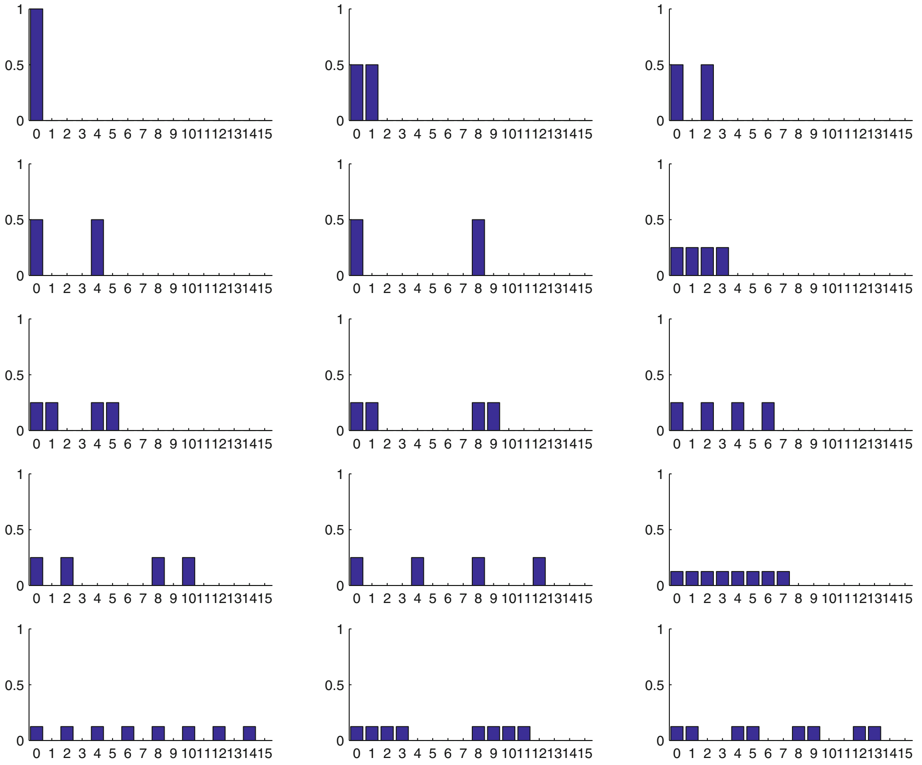

Fig. 7. Estimated probability distributions of (nibble-wise) faulty intermediate states for *Fault Model 1.* for all $c \in \mathbb{Z}_{16} \backslash \{15\}$.

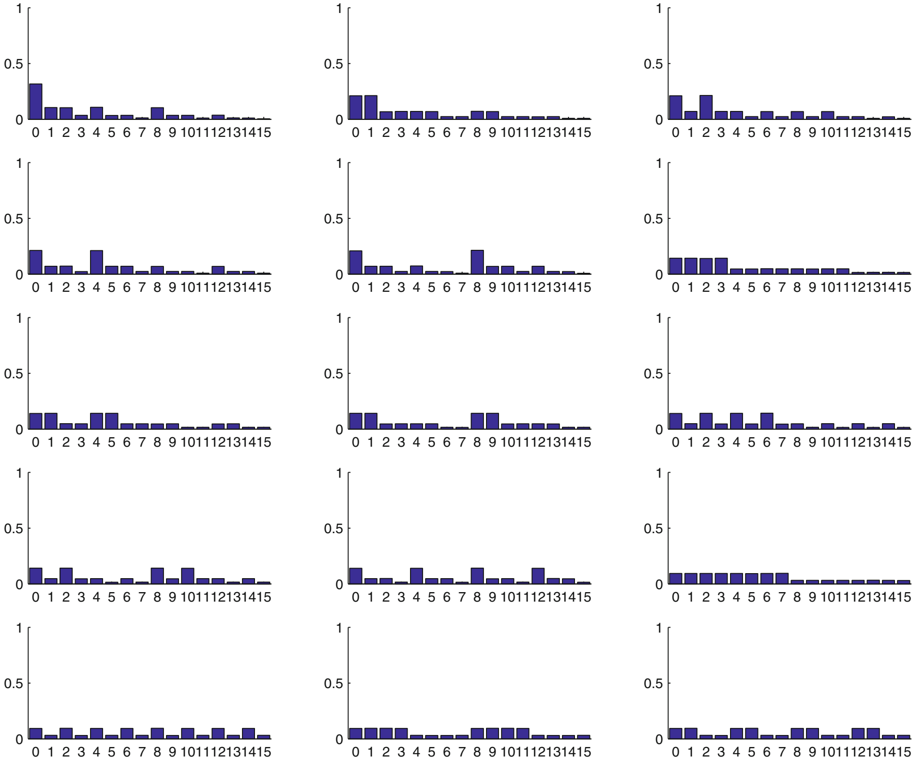

Fig. 8. Estimated probability distributions of (nibble-wise) faulty intermediate states for *Fault Model 2.* for all $c \in \mathbb{Z}_{16} \backslash \{15\}$.

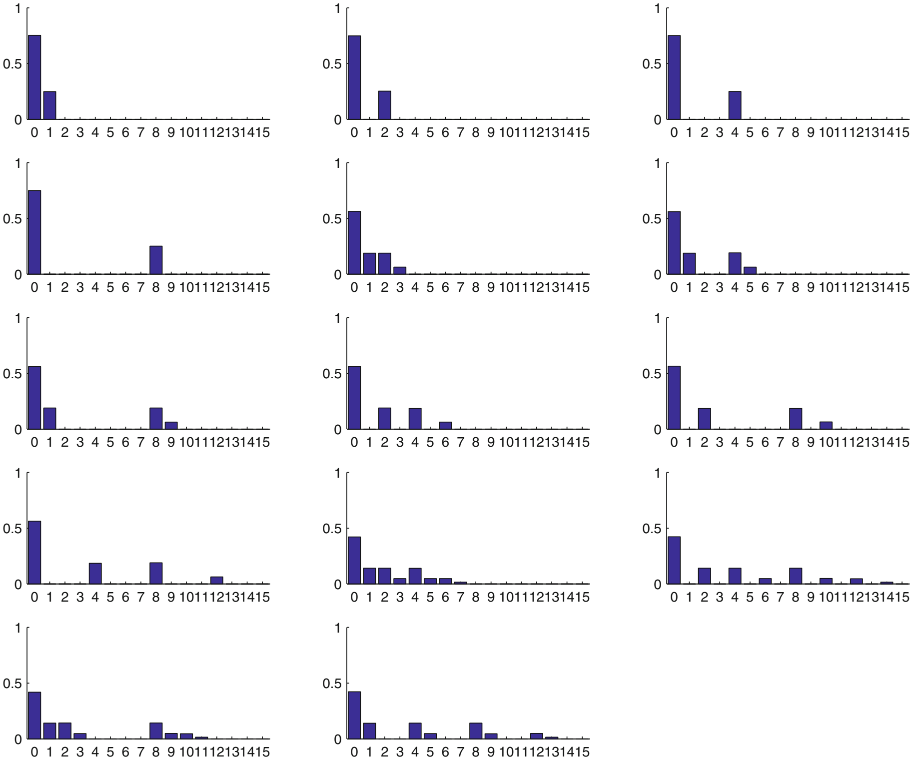

Fig. 9. Estimated probability distributions of (nibble-wise) faulty intermediate states for *Fault Model 3.* for all $c \in \mathbb{Z}_{16} \backslash \{0, 15\}$.

References

1. Akyildiz, I., Su, W., Sankarasubramaniam, Y., Cayirci, E.: A survey on sensor networks. IEEE Commun. Mag. **40**(8), 102–114 (2002)
2. Atzori, L., Iera, A., Morabito, G.: The internet of things: a survey. Comput. Netw. **54**(15), 2787–2805 (2010)
3. Avoine, G., Kara, O. (eds.): LightSec 2013. LNCS, vol. 8162. Springer, Heidelberg (2013)
4. Bagheri, N., Ebrahimpour, R., Ghaedi, N.: New differential fault analysis on present. EURASIP J. Adv. Signal Process. **2013**(1), 1–10 (2013). http://dx.doi.org/10.1186/1687-6180-2013-145
5. Balasch, J., Gierlichs, B., Verbauwhede, I.: An in-depth and black-box characterization of the effects of clock glitches on 8-bit MCUs. In: 2011 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), pp. 105–114, September 2011
6. Barenghi, A., Bertoni, G., Breveglieri, L., Pellicioli, M., Pelosi, G.: Low voltage fault attacks to aes. In: 2010 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST), pp. 7–12, June 2010
7. Barenghi, A., Hocquet, C., Bol, D., Standaert, F.-X., Regazzoni, F., Koren, I.: Exploring the feasibility of low cost fault injection attacks on sub-threshold devices through an example of a 65nm AES implementation. In: Juels, A., Paar, C. (eds.) RFIDSec 2011. LNCS, vol. 7055, pp. 48–60. Springer, Heidelberg (2012). http://dx.doi.org/10.1007/978-3-642-25286-0_4
8. Bassi, A., Horn, G.: Internet of things in 2020: A roadmap for the future. European Commission: Information Society and Media (2008)
9. Biehl, I., Meyer, B., Müller, V.: Differential fault attacks on elliptic curve cryptosystems. In: Bellare, M. (ed.) CRYPTO 2000. LNCS, vol. 1880, pp. 131–146. Springer, Heidelberg (2000). http://dx.doi.org/10.1007/3-540-44598-6_8
10. 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). http://dx.doi.org/10.1007/BFb0052259
11. 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://dx.doi.org/10.1007/978-3-540-74735-2_31
12. Boneh, D., DeMillo, R.A., Lipton, R.J.: On the importance of checking cryptographic protocols for faults. In: Fumy, W. (ed.) EUROCRYPT 1997. LNCS, vol. 1233, pp. 37–51. Springer, Heidelberg (1997). http://dx.doi.org/10.1007/3-540-69053-0_4
13. Fischer, W., Schmidt, J.M. (eds.): 2013 Workshop on Fault Diagnosis and Tolerance in Cryptography, Los Alamitos, CA, USA, 20 August 2013. IEEE (2013)
14. Fuhr, T., Jaulmes, E., Lomne, V., Thillard, A.: Fault attacks on aes with faulty ciphertexts only. In: 2013 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), pp. 108–118, August 2013
15. Gu, D., Li, J., Li, S., Ma, Z., Guo, Z., Liu, J.: Differential fault analysis on lightweight blockciphers with statistical cryptanalysis techniques. In: Bertoni, G., Gierlichs, B. (eds.) 2012 Workshop on Fault Diagnosis and Tolerance in Cryptography, Leuven, Belgium, 9 September 2012, pp. 27–33. IEEE (2012)
16. Guilley, S., Sauvage, L., Danger, J.L., Selmane, N.: Fault injection resilience. In: 2010 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), pp. 51–65, August 2010

17. Harpes, C., Kramer, G.G., Massey, J.L.: A generalization of linear cryptanalysis and the applicability of Matsui's piling-up lemma. In: Guillou, L.C., Quisquater, J.-J. (eds.) EUROCRYPT 1995. LNCS, vol. 921, pp. 24–38. Springer, Heidelberg (1995)
18. Hutter, M., Schmidt, J.M.: The temperature side channel and heating fault attacks. Cryptology ePrint Archive, Report 2014/190 (2014). http://eprint.iacr.org/
19. ISO: Information technology – security techniques – lightweight cryptography – part 2: Block ciphers. ISO/IEC 29192–2:2012, International Organization for Standardization, Geneva, Switzerland (2012)
20. Junod, P.: Statistical cryptanalysis of block ciphers. Ph.D. thesis, IC, Lausanne (2005)
21. Li, J., Gu, D.: Differential fault analysis on present. In: CHINACRYPT 2009, pp. 3–13 (2009)
22. Maistri, P.: Countermeasures against fault attacks: the good, the bad, and the ugly. In: Proceedings of the 2011 IEEE 17th International On-Line Testing Symposium, IOLTS 2011, p. 134137. IEEE Computer Society, Washington, DC (2011). http://dx.doi.org/10.1109/IOLTS.2011.5993825
23. Moro, N., Dehbaoui, A., Heydemann, K., Robisson, B., Encrenaz, E.: Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller. In: 2013 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC), pp. 77–88. IEEE (2013)
24. Mukhopadhyay, D.: An improved fault based attack of the advanced encryption standard. In: Preneel, B. (ed.) AFRICACRYPT 2009. LNCS, vol. 5580, pp. 421–434. Springer, Heidelberg (2009). http://dx.doi.org/10.1007/978-3-642-02384-2_26
25. Neve, M., Peeters, E., Samyde, D., Quisquater, J.J.: Memories: a survey of their secure uses in smart cards. In: Proceedings of the Second IEEE International Security in Storage Workshop, 2003, SISW 2003, pp. 62–62. IEEE (2003)
26. Piret, G., Quisquater, J.-J.: A differential fault attack technique against SPN structures, with application to the AES and KHAZAD. In: Walter, C.D., Koç, Ç.K., Paar, C. (eds.) CHES 2003. LNCS, vol. 2779, pp. 77–88. Springer, Heidelberg (2003). http://dx.doi.org/10.1007/978-3-540-45238-6_7
27. Rivain, M.: Differential fault analysis on DES middle rounds. In: Clavier, C., Gaj, K. (eds.) CHES 2009. LNCS, vol. 5747, pp. 457–469. Springer, Heidelberg (2009). http://dx.doi.org/10.1007/978-3-642-04138-9_32
28. Schmidt, J.M., Hutter, M.: Optical and em fault-attacks on crt-based rsa: concrete results. In: Karl C. Posch, J.W. (ed.) Austrochip 2007, 15th Austrian Workhop on Microelectronics, Proceedings, Graz, Austria, 11 October 2007, pp. 61–67. Verlag der Technischen Universität Graz (2007)
29. Schmidt, J.M., Hutter, M., Plos, T.: Optical fault attacks on aes: a threat in violet. In: Naccache, D., Oswald, E. (eds.) 6th Workshop on Fault Diagnosis and Tolerance in Cryptography - FDTC 2009, pp. 13–22. IEEE-CS Press (2009)
30. Skorobogatov, S.: Flash memory 'bumping' attacks. In: Mangard, S., Standaert, F.-X. (eds.) CHES 2010. LNCS, vol. 6225, pp. 158–172. Springer, Heidelberg (2010). http://dx.doi.org/10.1007/978-3-642-15031-9_11
31. 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). http://dx.doi.org/10.1007/978-3-642-01001-9_26

32. Wang, G., Wang, S.: Differential fault analysis on present key schedule. In: Proceedings of the 2010 International Conference on Computational Intelligence and Security, CIS 2010, pp. 362–366. IEEE Computer Society, Washington, DC (2010). http://dx.doi.org/10.1109/CIS.2010.84
33. Zhao, X., Guo, S., Wang, T., Zhang, F., Shi, Z.: Fault-propagate pattern based dfa on present and printcipher. Wuhan Univ. J. Nat. Sci. **17**(6), 485–493 (2012). http://dx.doi.org/10.1007/s11859-012-0875-7

Relating Undisturbed Bits to Other Properties of Substitution Boxes

Rusydi H. Makarim[1,2,3(✉)] and Cihangir Tezcan[3,4,5]

[1] Mathematical Institute, Leiden University, Leiden, The Netherlands
r.h.makarim@math.leidenuniv.nl
[2] CWI Cryptology Group, Amsterdam, The Netherlands
makarim@cwi.nl
[3] Institute of Applied Mathematics, Middle East Technical University, 06800 Çankaya, Ankara, Turkey
[4] Department of Mathematics, Middle East Technical University, 06800 Çankaya, Ankara, Turkey
[5] Institute of Informatics, CyDeS Cyber Defence and Security Laboratory, Middle East Technical University, 06800 Çankaya, Ankara, Turkey
cihangir@metu.edu.tr

Abstract. Recently it was observed that for a particular nonzero input difference to an S-Box, some bits in all the corresponding output differences may remain invariant. These specific invariant bits are called *undisturbed bits*. Undisturbed bits can also be seen as truncated differentials with probability 1 for an S-Box. The existence of undisturbed bits was found in the S-Box of PRESENT and its inverse. A 13-round improbable differential attack on PRESENT was provided by Tezcan and without using the undisturbed bits in the S-Box an attack of this type can only reach 7 rounds. Although the observation and the cryptanalytic application of undisturbed bits are given, their relation with other properties of an S-Box remain unknown. This paper presents some results on mathematical properties of S-Boxes having undisturbed bits. We show that an S-Box has undisturbed bits if any of its coordinate functions has a nontrivial linear structure. The relation of undisturbed bits with other cryptanalytic tools such as difference distribution table (DDT) and linear approximation table (LAT) are also given. We show that autocorrelation table is proven to be a more useful tool, compared to DDT, to obtain all nonzero input differences that yield undisturbed bits. Autocorrelation table can then be viewed as a counterpart of DDT for truncated differential cryptanalysis. Given an $n \times m$ balanced S-Box, we state that the S-Box has undisturbed bits whenever the degree of any of its coordinate function is quadratic.

Keywords: Block cipher · Substitution box · Undisturbed bits · Truncated differential

Cihangir Tezcan—The work of the second author was supported by The Scientific and Technological Research Council of Turkey (TÜBİTAK) under the grant 112E101 titled "Improbable Differential Cryptanalysis of Block Ciphers".

© Springer International Publishing Switzerland 2015
T. Eisenbarth and E. Öztürk (Eds.): LightSec 2014, LNCS 8898, pp. 109–125, 2015.
DOI: 10.1007/978-3-319-16363-5_7

1 Introduction

The emerging trends of small-scale computing devices raise the need for suitable cryptographic primitives, especially block ciphers. Two main challenges to design a block cipher for small-scale devices are the limited memory and available power. Some of the proposals for lightweight block ciphers, such as Present [2] and Rectangle [17], are designed in bit-oriented fashion. This is due to the efficiency of bit-level operation in hardware implementation.

In [16], Tezcan observed that for a particular nonzero input difference to the substitution box (S-Box) of Present, in all of the output differences, there exist some bits that remain the same. These specific invariant bits are called *undisturbed bits*. For instance, with input difference $\mathbf{9} = (1, 0, 0, 1)$ the least significant bit of every possible output difference is undisturbed and its value is equal to zero. The existence of undisturbed bits can also be equally seen as a truncated differential [7] with probability one for a given S-Box. This allows an attacker to have longer truncated differential for bit-oriented ciphers. In [16], a 13-round improbable differential attack was provided for Present and without using undisturbed bits, the best attack of this type can only reach 7 rounds (Table 1).

Table 1. The 4×4 S-Box of Present.

$\overline{x}$	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
$S(\overline{x})$	**12**	**5**	**6**	**11**	**9**	**0**	**10**	**13**	**3**	**14**	**15**	**8**	**4**	**7**	**1**	**2**

Proving the exact security bound of a block cipher against differential cryptanalysis is a challenging task. Typically the designer of block cipher would perform computer-aided search to find the best differential characteristic on reduced-round version of the cipher. One obvious way to improve the complexity of the searching algorithm is by reducing the search space. In [15] Sun *et al.* used the undisturbed bits in the S-Box of Present as additional constraint for searching the best differential in related-key settings. The existence of undisturbed bits remove some differential patterns that would never occur and, hence, reduce the search space of the differential characteristics. The undisturbed bits are then converted into linear inequalities for Mixed-Integer Linear Programming (MILP) model. The term *conditional differential propagation* is used by the authors to describe this behaviour.

In [16], it was shown that all 3×3 bijective S-Boxes contain undisturbed bits. Moreover, many 4×4 S-Boxes of cryptographic algorithms are also evaluated in [16], and it was observed that 66 % of these S-Boxes contain undisturbed bits. Since bit-oriented lightweight block ciphers use small S-Boxes, undisturbed bits pose a threat to the security of these ciphers.

Although previous literature have discussed the observation on undisturbed bits and its application in cryptanalysis of block ciphers, the relation of undisturbed bits with other properties of an S-Box remain unknown. The main goal

of this paper is to address this open problem and presents the relation of undisturbed bits to other properties in an S-Box. All necessary notations and preliminaries on Boolean functions and S-Boxes are given in Sect. 2.

We breakdown the primary goal of this paper into several sub-problems. The first sub-problem is, one may ask the implication of undisturbed bits to the component functions of an S-Box. Specifically, we would like to focus on the component functions of an S-Box where the undisturbed bits occur. The second sub-problem is the relation of undisturbed bits with other cryptanalytic tools for S-Boxes. We want to see the existence of undisturbed bits from the point of view of two well-known cryptanalytic tools, *difference distribution table* (DDT) [1] and *linear approximation table* (LAT) [10]. We will address these two sub-problems and show the relation of undisturbed bits with the notion of linear structure in Sect. 3. The third sub-problem in this work deals with a problem of developing dedicated cryptanalytic tool to obtain all nonzero input differences that yield undisturbed bits. In Sect. 4 autocorrelation table will be introduced as a cryptanalytic tool, in addition to DDT and LAT, that can be used to find undisturbed bits. Lastly, we ask what would be the property of an S-Box that may indicate whether an S-Box has undisturbed bits. We will show in Sect. 5 that a balanced $n \times m$ S-Box with a quadratic coordinate function has undisturbed bits. We conclude this paper in Sect. 6.

2 Notations and Preliminaries

The cardinality of a set V is denoted by $|V|$. Let $\mathbb{F}_2 = \{0, 1\}$ be a finite field with two elements and $\mathbb{F}_2^n$ be n-dimensional vector space over $\mathbb{F}_2$. Any element of $\mathbb{F}_2^n$ is denoted by $\overline{x} = (x_{n-1}, \dots, x_0)$. The notation $\oplus$ is used to denote the addition in $\mathbb{F}_2$ as well as $\mathbb{F}_2^n$. The vector $\overline{x} = (x_{n-1}, \dots, x_0) \in \mathbb{F}_2^n$ can be represented as integer by $\boldsymbol{x} = \sum_{i=0}^{n-1} x_i 2^i$ and its associated integer representation is written using boldface type font. The standard basis for $\mathbb{F}_2^n$ is represented by

$$\overline{e}_{n-1} = (1, 0, 0, \dots, 0), \quad \dots \quad \overline{e}_1 = (0, \dots, 0, 1, 0), \quad \overline{e}_0 = (0, 0, \dots, 0, 1)$$

The vector $\overline{e}_i$ is called the i-th *standard basis* of $\mathbb{F}_2^n$. The integer representation of each i-th standard basis of $\mathbb{F}_2^n$ is given by $\mathbf{2}^i$. The *inner product* of vectors $\overline{x}, \overline{y} \in \mathbb{F}_2^n$ is defined as $\overline{x} \cdot \overline{y} = x_{n-1}y_{n-1} \oplus \cdots \oplus x_0 y_0$. The *weight* of vector $\overline{x} \in \mathbb{F}_2^n$ is defined as the number of its nonzero components, denoted $\mathrm{wt}(\overline{x})$. Note that in this paper every vector is considered as column vector, but we will continue writing it in row-wise manner.

2.1 Boolean Functions

A *Boolean function* $f : \mathbb{F}_2^n \mapsto \mathbb{F}_2$ is a map from $\mathbb{F}_2^n$ to $\mathbb{F}_2$. The associated *sign function* $\widehat{f}(\overline{x})$ for every Boolean function f is defined by $\widehat{f}(\overline{x}) = (-1)^{f(\overline{x})} \in \{-1, 1\}$. The *weight* of a Boolean function f, denoted by $\mathrm{wt}(f)$, is defined as $\mathrm{wt}(f) = |\{\overline{x} \in \mathbb{F}_2^n \mid f(\overline{x}) \neq 0\}|$. A Boolean function f with $\mathrm{wt}(f) = 2^{n-1}$ is

called a *balanced function*. If for every $\overline{x} \in \mathbb{F}_2^n$ the Boolean function $f(\overline{x}) = \tau$ for a fixed $\tau \in \mathbb{F}_2$, then we call f a *constant function*. The *distance* of two Boolean functions f, g, denoted by $\mathrm{dt}(f, g)$ is defined as the number of entry in which they differ, i.e. $\mathrm{dt}(f, g) = |\{\overline{x} \in \mathbb{F}_2^n \mid f(\overline{x}) \neq g(\overline{x})\}|$.

A Boolean function can be represented using algebraic expression

$$f(\overline{x}) = f(x_{n-1}, \ldots, x_1, x_0) = \bigoplus_{\overline{u} \in \mathbb{F}_2^n} a_{\overline{u}} x_{n-1}^{u_{n-1}} \cdots x_0^{u_0} = \bigoplus_{\overline{u} \in \mathbb{F}_2^n} a_{\overline{u}} \overline{x}^{\overline{u}} \tag{1}$$

The coefficient $a_{\overline{u}}$ is obtained by $a_{\overline{u}} = \bigoplus_{\overline{x} \preceq \overline{u}} f(\overline{x})$ where $\overline{x} \preceq \overline{u}$ means that $x_i \leq u_i$ for all $0 \leq i \leq n-1$ (we say that $\overline{u}$ *covers* $\overline{x}$). We refer to expression given in Eq. (1) as the *algebraic normal form* (ANF) of f. The *degree* of Boolean function, $\deg(f)$, is defined as the maximal monomial degree in its ANF representation. The following proposition gives an upper bound of the degree for balanced function.

Proposition 1 [14]**.** *For a balanced n-variable Boolean function with $n \geq 2$, $\deg(f) \leq n-1$.*

An *affine function* is a Boolean function such that its ANF is of the form $\overline{\omega} \cdot \overline{x} \oplus \epsilon = \omega_{n-1} x_{n-1} \oplus \cdots \oplus \omega_0 x_0 \oplus \epsilon$ for $\overline{\omega} = (\omega_{n-1}, \ldots, \omega_0) \in \mathbb{F}_2^n$ and $\epsilon \in \mathbb{F}_2$. The vector $\overline{\omega}$ is the *coefficient vector* of the affine function. If $\epsilon = 0$, the function $\overline{\omega} \cdot \overline{x}$ is called a *linear function*. The following proposition characterizes the weight of affine functions.

Proposition 2. *Every affine function with nonzero coefficient vector is balanced. If the coefficient vector is zero vector, the affine function is a constant function.*

In the analysis of a Boolean function, *Walsh-Hadamard Transform* is an important tool that could determine various properties of the function. We give the following definition of Walsh-Hadamard Transform as well as its inverse transform.

Definition 1 (Walsh-Hadamard Transform). *The Walsh value of f at $\overline{\omega} \in \mathbb{F}_2^n$ is defined by*

$$\mathcal{W}_f(\overline{\omega}) = \sum_{\overline{x} \in \mathbb{F}_2^n} (-1)^{f(\overline{x})} (-1)^{\overline{\omega} \cdot \overline{x}} = \sum_{\overline{x} \in \mathbb{F}_2^n} \widehat{f}(\overline{x}) (-1)^{\overline{\omega} \cdot \overline{x}}$$

The inverse transform is defined by

$$\widehat{f}(\overline{x}) = 2^{-n} \sum_{\overline{\omega} \in \mathbb{F}_2^n} \mathcal{W}_f(\overline{\omega}) (-1)^{\overline{x} \cdot \overline{\omega}}$$

The vector $(\mathcal{W}_f(\mathbf{0}), \ldots, \mathcal{W}_f(\mathbf{2}^n - \mathbf{1}))$ is called the *Walsh spectrum* of f. One of the properties of a Boolean function that can be determined from the Walsh value is balancedness.

Proposition 3. *The Boolean function f is balanced if and only if $\mathcal{W}_f(\overline{0}) = 0$.*

Another important tool in analysis of Boolean functions is the notion of *auto-correlation* and its relation with undisturbed bits are discussed in Sect. 3.

Definition 2 (Autocorrelation). *The autocorrelation of n-variable Boolean function f at $\overline{\alpha} \in \mathbb{F}_2^n$ is defined by*

$$r_f(\overline{\alpha}) = \sum_{\overline{x} \in \mathbb{F}_2^n} (-1)^{f(\overline{x})}(-1)^{f(\overline{x} \oplus \overline{\alpha})} = \sum_{\overline{x} \in \mathbb{F}_2^n} (-1)^{f(\overline{x}) \oplus f(\overline{x} \oplus \overline{\alpha})}.$$

We refer to vector $(r_f(\mathbf{0}), \ldots, r_f(\mathbf{2}^n - \mathbf{1}))$ as the *autocorrelation spectrum* of f. The relation of autocorrelation and Walsh-transform is given by the Wiener-Khinthcine's Theorem.

Theorem 1 (Wiener-Khinthcine [12]). *The expression of the autocorrelation in terms of Walsh value is equal to*

$$r_f(\overline{\alpha}) = 2^{-n} \sum_{\overline{\omega} \in \mathbb{F}_2^n} \mathcal{W}_f^2(\overline{\omega})(-1)^{\overline{\alpha} \cdot \overline{\omega}}$$

A cryptographic criteria which is closely related to its autocorrelation is *Strict Avalanche Criterion* (SAC). An n-variable Boolean function f satisfies SAC if changing any one of the n bits in the input results in the output of the function being changed with probability 1/2. It is clear that the following proposition follows from the definition of SAC and could be treated as an equivalent definition.

Proposition 4. *An n-variable Boolean function f satisfies SAC if and only if the function $f(\overline{x}) \oplus f(\overline{x} \oplus \overline{\alpha})$ is balanced for every $\overline{\alpha} \in \mathbb{F}_2^n$ with $wt(\overline{\alpha}) = 1$. Equivalently, the function f satisfies SAC if and only if $r_f(\overline{\alpha}) = 0$, with $wt(\overline{\alpha}) = 1$.*

An n-variable Boolean function is said to satisfy *propagation criterion* of degree k, which we denote by PC(k), if changing any i ($1 \leq i \leq k$) of the n bits in the input results in the output of the function being changed for half of the times. This definition generalizes the notion of SAC, which clearly equals to PC(1) function. The following proposition is analogous to the one given in Proposition 4.

Proposition 5. *An n-variable Boolean function f satisfies PC(k) if and only if all of the given values*

$$r_f(\overline{\alpha}) = \sum_{\overline{x} \in \mathbb{F}_2^n} (-1)^{f(\overline{x})}(-1)^{f(\overline{x} \oplus \overline{\alpha})} = 0 \qquad 1 \leq wt(\overline{\alpha}) \leq k$$

The *derivative* of f at $\overline{\alpha} \in \mathbb{F}_2^n$ is defined as $D_{\overline{\alpha}} f(\overline{x}) = f(\overline{x}) \oplus f(\overline{x} \oplus \overline{\alpha})$. The derivative of f at any point in $\mathbb{F}_2^n$ can also be treated as an n-variable Boolean function. The autocorrelation of a Boolean function can then be expressed in terms of its derivative as $r_f(\overline{\alpha}) = \sum_{\overline{x} \in \mathbb{F}_2^n} (-1)^{D_{\overline{\alpha}} f(\overline{x})}$. The following proposition gives an upper bound of the degree of a derivative function.

Proposition 6 [9]**.** *If f is an n-variable Boolean function and $\overline{\alpha} \in \mathbb{F}_2^n$, then* $\deg(D_{\overline{\alpha}} f) \leq \deg(f) - 1$.

If $D_{\overline{\alpha}} f(\overline{x})$ is a constant function, then $\overline{\alpha}$ is a *linear structure* of f [6,8]. The zero vector $\overline{0}$ is a trivial linear structure since $D_{\overline{0}} f(\overline{x}) = 0$ for all $\overline{x} \in \mathbb{F}_2^n$. We say that the function f has a linear structure if there exists a nonzero vector $\overline{\alpha} \in \mathbb{F}_2^n$ such that $D_{\overline{\alpha}} f(\overline{x})$ is a constant function. The notation $\mathcal{LS}_f$ is used to denote the set of all linear structures of f. The set of all n-variable Boolean functions that has linear structure is denoted by $\mathcal{LS}(n)$. From the point of view of autocorrelation, a vector in $\mathbb{F}_2^n$ is a linear structure if it satisfies the following proposition.

Proposition 7. *The vector $\overline{\alpha} \in \mathbb{F}_2^n$ is a linear structure of f if and only if $r_f(\overline{\alpha}) = \pm 2^n$.*

Proposition 8. *Any vector in $\mathbb{F}_2^n$ is a linear structure of every affine functions.*

Proof. Let $\overline{\alpha} \in \mathbb{F}_2^n$. Recall that we can represent affine function as $\overline{\omega} \cdot \overline{x} \oplus \epsilon$ with $\overline{\omega} \in \mathbb{F}_2^n$ and $\epsilon \in \mathbb{F}_2$. The derivative of affine function $\overline{\omega} \cdot \overline{x} \oplus \epsilon$ at $\overline{\alpha}$ is equal to

$$\begin{aligned}(\overline{\omega} \cdot \overline{x} \oplus \epsilon) \oplus (\overline{\omega} \cdot (\overline{x} \oplus \overline{\alpha}) \oplus \epsilon) &= (\overline{\omega} \cdot \overline{x} \oplus \epsilon) \oplus ((\overline{\omega} \cdot \overline{x} \oplus \overline{\omega} \cdot \overline{\alpha}) \oplus \epsilon) \\ &= \overline{\omega} \cdot \overline{\alpha}\end{aligned}$$

This implies that the derivative of affine function $\overline{\omega} \cdot \overline{x} \oplus \epsilon$ at $\overline{\alpha}$ is equal to $\overline{\omega} \cdot \overline{\alpha}$ for all $\overline{x} \in \mathbb{F}_2^n$ and, hence, is a constant function. Clearly $\overline{\alpha}$ is a linear structure of $\overline{\omega} \cdot \overline{x} \oplus \epsilon$. □

2.2 Substitution Boxes

An $n \times m$ S-Box is a mapping $S : \mathbb{F}_2^n \mapsto \mathbb{F}_2^m$. The internal structure of an S-Box can be decomposed into Boolean functions. Let $\overline{y} = (y_{m-1}, \ldots, y_0) \in \mathbb{F}_2^m$ and $\overline{y} = S(\overline{x})$. The component of $\overline{y}$ can be computed by $y_i = h_i(\overline{x})$. The function $h_i : \mathbb{F}_2^n \mapsto \mathbb{F}_2$ is called the *coordinate function* of S-Box S. The *component functions* of S-Box S are the mapping $\overline{b} \cdot S(\overline{x})$ for all nonzero $\overline{b} \in \mathbb{F}_2^m$. The component functions are essentially generalization of coordinate functions of an S-Box by considering its linear combination. It follows that the coordinate function $h_i(\overline{x}) = \overline{e}_i \cdot S(\overline{x})$ where $\overline{e}_i$ is the i-th standard basis of $\mathbb{F}_2^m$.

An $n \times m$ S-Box S is *balanced* if it takes every value of $\mathbb{F}_2^m$ the same number 2^{n-m} of times [3]. The following proposition characterizes a balanced S-Box from the balancedness of its component functions.

Proposition 9 [3]**.** *An $n \times m$ S-Box is balanced if and only if its component functions are balanced, that is if and only if for every nonzero $\overline{b} \in \mathbb{F}_2^m$, the Boolean function $\overline{b} \cdot S(\overline{x})$ is balanced.*

The notion of linear structures in Boolean functions can be extended for the case of S-Boxes. The definition of an S-Box that has a linear structure was originally proposed by Chaum [5] and Evertse [6]. They define that an S-Box has a linear structure by considering the existence of nontrivial linear structure in any of the component functions of the S-Box.

Definition 3 (S-Box with linear structures [5,6,11]**).** *An $n \times m$ S-Box S is said to have a linear structure if there exists a nonzero vector $\overline{\alpha} \in \mathbb{F}_2^n$ together with a nonzero vector $\overline{b} \in \mathbb{F}_2^m$ such that $\overline{b} \cdot S(\overline{x}) \oplus \overline{b} \cdot S(\overline{x} \oplus \overline{\alpha})$ takes the same value $c \in \mathbb{F}_2$ for all $\overline{x} \in \mathbb{F}_2^n$.*

Proposition 10. *An $n \times m$ S-Box S is said to have a linear structure if there exists a nonzero vector $\overline{\alpha} \in \mathbb{F}_2^n$ together with a nonzero vector $\overline{b} \in \mathbb{F}_2^m$ such that $r_{\overline{b} \cdot S}(\overline{\alpha}) = \pm 2^n$*

In the cryptanalysis of block ciphers, the two most well-known cryptanalytic tools to analyse properties of an S-Box are DDT and LAT.

Let $\overline{x}, \overline{x}' \in \mathbb{F}_2^n$ be two inputs to the S-Box S and $\overline{y} = S(\overline{x})$, $\overline{y}' = S(\overline{x}')$ be their corresponding outputs. We refer to the difference in the input $\overline{x} \oplus \overline{x}' = \overline{\alpha}$ as the *input difference* to S. Similarly $\overline{y} \oplus \overline{y}' = \overline{\beta}$ is the *output difference* of S corresponding to input difference $\overline{\alpha}$. DDT examines how many times a certain output difference of an S-Box occur for a given input difference. The definition of DDT is given as follows.

Definition 4. *For an $n \times m$ S-Box S, the entry in the row $\overline{s} \in \mathbb{F}_2^n$ and column $\overline{t} \in \mathbb{F}_2^m$ (considering their integer representation) of difference distribution table of S is defined by $\mathsf{DDT}(\boldsymbol{s}, \boldsymbol{t}) = |\{\overline{x} \in \mathbb{F}_2^n \mid S(\overline{x}) \oplus S(\overline{x} \oplus \overline{s}) = \overline{t}\}|$.*

The probability of an input difference $\overline{\alpha}$ that yields the output difference $\overline{\beta}$ is then defined by

$$\begin{aligned}\mathbf{Pr}_S[\overline{\alpha} \rightarrow \overline{\beta}] &= 2^{-n}|\{\overline{x} \in \mathbb{F}_2^n \mid S(\overline{x}) \oplus S(\overline{x} \oplus \overline{\alpha}) = \overline{\beta}\}| \\ &= 2^{-n} \cdot \mathsf{DDT}(\boldsymbol{\alpha}, \boldsymbol{\beta})\end{aligned}$$

On the other hand, LAT is used to find the best linear approximation for an S-Box involving the parity bits of its input and output. The definition of linear approximation table is given as follows.

Definition 5. *For an $n \times m$ S-Box S, the linear approximation table of S at row $\overline{s} \in \mathbb{F}_2^n$ and column $\overline{t} \in \mathbb{F}_2^m$ (considering their integer representation) is defined as*

$$\mathsf{LAT}(\boldsymbol{s}, \boldsymbol{t}) = |\{\overline{x} \in \mathbb{F}_2^n \mid \overline{s} \cdot \overline{x} = \overline{t} \cdot S(\overline{x})\}| - 2^{n-1}$$

3 Undisturbed Bits and Linear Structures

In this section we recall the definition of undisturbed bits and provide its relations with autocorrelation, derivative, and linear structure of coordinate functions in an S-Box. The notation $S = (h_{m-1}, \ldots, h_0)$ will be used consistently for the rest of the paper to denote the $n \times m$ S-Box $S : \mathbb{F}_2^n \mapsto \mathbb{F}_2^m$ with coordinate functions $h_{m-1}, \ldots, h_0$, where $h_i : \mathbb{F}_2^n \mapsto \mathbb{F}_2$.

Definition 6 (Undisturbed Bits). *Let $\overline{\alpha} \in \mathbb{F}_2^n$ be a nonzero input difference to S-Box S and $\Omega_{\overline{\alpha}} = \{\overline{\beta} = (\beta_{m-1}, \ldots, \beta_0) \in \mathbb{F}_2^m \mid \mathbf{Pr}_S[\overline{\alpha} \rightarrow \overline{\beta}] > 0\}$ be the*

set of all possible output differences of S corresponding to $\overline{\alpha}$. If $\beta_i = c$ for a fixed $c \in \mathbb{F}_2$ and for all $\overline{\beta} \in \Omega_{\overline{\alpha}}$ with $i \in \{0, \ldots, m-1\}$, then the S-Box S has undisturbed bits. In particular, we say that for input difference $\overline{\alpha}$, the i-th bit of the output difference of S is undisturbed (and its value is c).

Recall that any output of the S-Box as the element of $\mathbb{F}_2^m$ can be computed component-wisely using coordinate functions of an S-Box. If $\mathbf{Pr}_S[\overline{\alpha} \to \overline{\beta}] > 0$, then there exists a $\overline{v} \in \mathbb{F}_2^n$ such that $S(\overline{v}) \oplus S(\overline{v} \oplus \overline{\alpha}) = \overline{\beta}$. It follows that the component of the output difference vectors $\overline{\beta} = (\beta_{m-1}, \ldots, \beta_0)$ can be obtained by $\beta_i = h_i(\overline{v}) \oplus h_i(\overline{v} \oplus \overline{\alpha})$. The following result is an implication from this observation.

Theorem 2. *For a nonzero input difference $\overline{\alpha} \in \mathbb{F}_2^n$ and $i \in \{0, \ldots, m-1\}$, the i-th bit of the output difference of S is undisturbed if and only if $D_{\overline{\alpha}} h_i(\overline{x}) = h_i(\overline{x}) \oplus h_i(\overline{x} \oplus \overline{\alpha})$ is a constant function.*

Proof. Suppose for an input difference $\overline{\alpha}$ the i-th bit of the output difference of S is undisturbed. Let $\Omega_{\overline{\alpha}} = \{\overline{\beta} = (\beta_{m-1}, \ldots, \beta_0) \in \mathbb{F}_2^m \mid \mathbf{Pr}_S[\overline{\alpha} \to \overline{\beta}] > 0\}$ be the set of all possible output differences of S corresponding to $\overline{\alpha}$. Definition 6 tells us that for all $\overline{\beta} = (\beta_{m-1}, \ldots, \beta_0) \in \Omega_{\overline{\alpha}}$ the component $\beta_i = c$ for a fixed $c \in \mathbb{F}_2$. Since $\beta_i = h_i(\overline{v}) \oplus h_i(\overline{v} \oplus \overline{\alpha})$ for some $\overline{v} \in \mathbb{F}_2^n$ and because the computation of output differences in $\Omega_{\overline{\alpha}}$ run through all the elements of $\mathbb{F}_2^n$, clearly $D_{\overline{\alpha}} h_i(\overline{x}) = h_i(\overline{x}) \oplus h_i(\overline{x} \oplus \overline{\alpha}) = c$ for all $\overline{x} \in \mathbb{F}_2^n$. Hence $D_{\overline{\alpha}} h_i(\overline{x})$ is a constant function. The converse part of the proof can be done by reversing the previous step. □

The value of undisturbed bits can then be deduced whether the constant function $D_{\overline{\alpha}} h_i(\overline{x})$ is equal to zero or one, for each $\overline{x} \in \mathbb{F}_2^n$. Because $D_{\overline{\alpha}} h_i(\overline{x})$ is a constant function, then the nonzero vector $\overline{\alpha}$ is a linear structure of the coordinate function h_i. Equivalently, since $\overline{\alpha}$ is a nonzero vector, then h_i is a function with linear structure. This result shows that a particular S-Box has undisturbed bits if any of its coordinate functions has a nontrivial linear structure. In order to see if an S-Box has undisturbed bits, it is sufficient to check the derivative of each coordinate function at every nonzero element of $\mathbb{F}_2^n$.

Theorem 2 also relates an S-Box which has undisturbed bits with Definition 3 about an S-Box with linear structures. It shows that an S-Box that has undisturbed bits belongs to special class of S-Boxes with linear structures by only considering the existence of linear structures in its coordinate functions. This can be described by the following proposition, and it can be treated as an equivalent definition for an S-Box that has undisturbed bits.

Proposition 11. *An $n \times m$ S-Box S is said to have an undisturbed bit if there exists a nonzero vector $\overline{\alpha} \in \mathbb{F}_2^n$ together with a nonzero vector $\overline{b} \in \mathbb{F}_2^m$ with $wt(b) = 1$ such that $\overline{b} \cdot S(\overline{x}) \oplus \overline{b} \cdot S(\overline{x} \oplus \overline{\alpha})$ takes the same value $c \in \mathbb{F}_2$ for all $\overline{x} \in \mathbb{F}_2^n$.*

In other words, if an S-Box S has undisturbed bits, then S has a linear structure. However, the converse is not true in general. Thus, Definition 3 can be seen as a generalization of undisturbed bits.

The existence of undisturbed bits in an S-Box may also be used to describe the unsatisfiability of the corresponding coordinate functions against SAC. We state it in the following remark.

Remark 1. Let $\mathcal{I}_i = \{\overline{\alpha} \in \mathbb{F}_2^n,\ \overline{\alpha} \neq \overline{0} \mid h_i(\overline{x}) \oplus h_i(\overline{x} \oplus \overline{\alpha}) \text{ is a constant function}\}$ be the set such that for any $\overline{\alpha} \in \mathcal{I}_i$ the i-th bit of the output difference of S is undisturbed. Equivalently $\mathcal{I}_i$ is the set of all nonzero linear structures of the coordinate function h_i, i.e. $\mathcal{I}_i = \mathcal{LS}_{h_i} \setminus \{\overline{0}\}$. We set

$$d = \min_{\overline{\alpha} \in \mathcal{I}_i} \mathrm{wt}(\overline{\alpha})$$

If $d = 1$, then from Proposition 4 it follows that the coordinate function h_i does not satisfy Strict Avalanche Criterion (SAC). However, this remark can not be generalized for $d > 1$. The reason is because if there exists a d' with $1 \leq d' < d$ such that the coordinate function does not satisfy PC(d') then d is not a proper bound for the unsatisfiability condition.

A trivial lemma can be derived from Theorem 2 to indicate whether an S-Box has undisturbed bits from the autocorrelation of its coordinate functions. We will use the following lemma to show the relation of other cryptanalytic tools with undisturbed bits.

Lemma 1. *For a nonzero input difference $\overline{\alpha} \in \mathbb{F}_2^n$, the i-th bit of the output difference of S is undisturbed if and only if*

$$r_{h_i}(\overline{\alpha}) = \pm 2^n$$

for $i \in \{0, \ldots, m-1\}$.

Proof. Suppose for a nonzero input difference $\overline{\alpha} \in \mathbb{F}_2^n$, the i-th bit of the output difference of S is undisturbed. From Theorem 2 the vector $\overline{\alpha}$ is a linear structure of coordinate function h_i. It follows that from Proposition 7 we have $r_{h_i}(\overline{\alpha}) = \pm 2^n$. The converse can be proven by reversing the previous steps. $\square$

The remaining part of this section describes the relation of some existing cryptanalytic tools with undisturbed bits. In particular, we give the relation of undisturbed bits with two most important cryptanalytic tools for an S-Box, namely DDT and LAT. The following theorem of [18] provides a relation between DDT and the autocorrelation of the component functions of an S-Box.

Theorem 3 [18]**.** *The relation between difference distribution table and the autocorrelation of the component functions of S is given by*

$$r_{\overline{j} \cdot S}(\overline{\alpha}) = \sum_{\overline{v} \in \mathbb{F}_2^m} \mathsf{DDT}(\boldsymbol{\alpha}, \boldsymbol{v})(-1)^{\overline{j} \cdot \overline{v}}$$

for $\overline{\alpha} \in \mathbb{F}_2^n$ and $\overline{j} \in \mathbb{F}_2^m$.

Using Lemma 1 the relation of undisturbed bits and DDT can be easily shown in Corollary 1.

Corollary 1 (DDT and Undisturbed Bits). *For a nonzero input difference $\overline{\alpha} \in \mathbb{F}_2^n$, the i-th bit of the output difference of S is undisturbed if and only if*

$$\sum_{\overline{v} \in \mathbb{F}_2^m} \mathsf{DDT}(\boldsymbol{\alpha}, \boldsymbol{v})(-1)^{\overline{e_i} \cdot \overline{v}} = \pm 2^n$$

for $i \in \{0, \ldots, m-1\}$ and $\overline{e}_i$ is the i-th standard basis of $\mathbb{F}_2^m$.

Proof. Suppose for a nonzero input difference $\overline{\alpha} \in \mathbb{F}_2^n$, the i-th bit of the output difference of S is undisturbed. From Lemma 1 we have $r_{h_i}(\overline{\alpha}) = \pm 2^n$. Since $r_{h_i}(\overline{\alpha}) = r_{\overline{e}_i \cdot S}(\overline{\alpha})$ it follows from Theorem 3 that $\sum_{\overline{v} \in \mathbb{F}_2^m} \mathsf{DDT}(\boldsymbol{\alpha}, \boldsymbol{v})(-1)^{\overline{e_i} \cdot \overline{v}} = \pm 2^n$. The converse can be trivially proved by reversing the previous steps. □

Linear approximation table (LAT) is used as a counterpart of DDT in the domain of linear cryptanalysis. Although undisturbed bits are useful in constructing truncated differential for bit-oriented cipher, one may also indicate the existence of undisturbed bits from LAT. We will use a well-known relation of LAT and the Walsh value of component functions of an S-Box in Lemma 2. Together with Theorem 1 (Wiener-Khintchine) and Lemma 1, the relation of LAT and undisturbed bits can be established. The main result is given in Theorem 4.

Lemma 2. *The relation between linear approximation table of S and the Walsh transform of the component functions of S is given by*

$$\mathsf{LAT}(\boldsymbol{a}, \boldsymbol{b}) = \frac{1}{2} \mathcal{W}_{\overline{b} \cdot S}(\overline{a})$$

for $\overline{a} \in \mathbb{F}_2^n$ and $\overline{b} \in \mathbb{F}_2^m$.

Theorem 4 (LAT and Undisturbed Bits). *For a nonzero input difference $\overline{\alpha} \in \mathbb{F}_2^n$, the i-th bit of the output difference of S is undisturbed if and only if*

$$2^{2-n} \sum_{\overline{a} \in \mathbb{F}_2^n} \mathsf{LAT}(\boldsymbol{a}, \boldsymbol{2^i})^2 (-1)^{\overline{\alpha} \cdot \overline{a}} = \pm 2^n$$

for $i \in \{0, \ldots, m-1\}$.

Proof. Firstly, we claim that $2^{2-n} \sum_{\overline{a} \in \mathbb{F}_2^n} \mathsf{LAT}(\boldsymbol{a}, \boldsymbol{b})^2 (-1)^{\overline{\alpha} \cdot \overline{a}} = r_{\overline{b} \cdot S}(\overline{\alpha})$. The proof of the claim is as follows

$$\begin{aligned} 2^{2-n} \sum_{\overline{a} \in \mathbb{F}_2^n} \mathsf{LAT}(\boldsymbol{a}, \boldsymbol{b})^2 (-1)^{\overline{\alpha} \cdot \overline{a}} &= 2^{-n} \sum_{\overline{a} \in \mathbb{F}_2^n} 2^2 \cdot \mathsf{LAT}(\boldsymbol{a}, \boldsymbol{b})^2 (-1)^{\overline{\alpha} \cdot \overline{a}} \\ &= 2^{-n} \sum_{\overline{a} \in \mathbb{F}_2^n} (2 \cdot \mathsf{LAT}(\boldsymbol{a}, \boldsymbol{b}))^2 (-1)^{\overline{\alpha} \cdot \overline{a}} \\ &= 2^{-n} \sum_{\overline{a} \in \mathbb{F}_2^n} \mathcal{W}_{\overline{b} \cdot S}(\overline{a})^2 (-1)^{\overline{\alpha} \cdot \overline{a}} && \text{from Lemma 2} \\ &= r_{\overline{b} \cdot S}(\overline{\alpha}) && \text{from Theorem 1} \end{aligned}$$

Clearly we have

$$2^{2-n} \sum_{\overline{a} \in \mathbb{F}_2^n} \mathsf{LAT}(\boldsymbol{a}, \boldsymbol{2^i})^2 (-1)^{\overline{\alpha} \cdot \overline{a}} = r_{\overline{e}_i \cdot S}(\overline{\alpha}) = r_{h_i}(\overline{\alpha}) = \pm 2^n$$

where $\overline{e}_i$ is the i-th standard basis of $\mathbb{F}_2^m$. Immediately from Lemma 1, for nonzero input difference $\overline{\alpha}$ the i-th bit of the output difference of S is undisturbed.

Conversely, if for a nonzero input difference $\overline{\alpha}$ the i-th bit of the output difference of S is undisturbed, Lemma 1 implies that $r_{h_i}(\overline{\alpha}) = \pm 2^n$. From our claim we can have $\pm 2^n = r_{\overline{e_i} \cdot S}(\overline{\alpha}) = 2^{2-n} \sum_{\overline{a} \in \mathbb{F}_2^n} \mathsf{LAT}(\boldsymbol{a}, \boldsymbol{2^i})^2 (-1)^{\overline{\alpha} \cdot \overline{a}}$. □

4 Autocorrelation Table

One way to check the existence of undisturbed bits in an S-Box is by taking a nonzero input difference and see whether there are some bits in all the corresponding output differences that remain invariant. This can be done by observing the DDT of an S-Box. However, this indirect approach can be improved if one is able to find a dedicated cryptanalytic tool for the case of undisturbed bits.

In this section, we extend the result of Lemma 1 and provide a tool called *autocorrelation table*, which was also appeared previously in [18]. Though it was introduced earlier, the application of autocorrelation table for cryptanalysis of block ciphers was not mentioned. We will show that autocorrelation table is proven to be a more useful tool, compared to DDT, to check if an S-Box has undisturbed bits. Moreover, we will be able to obtain all nonzero input differences that has undisturbed bits in its corresponding output differences. Because undisturbed bit is also a truncated differential of probability one in an S-Box, autocorrelation table can be viewed as a counterpart of DDT in the domain of truncated differential cryptanalysis.

Definition 7 (Autocorrelation Table [18]). *For $\overline{a} \in \mathbb{F}_2^n$ and $\overline{b} \in \mathbb{F}_2^m$, we define autocorrelation table of S-Box S, denoted as ACT, where the entry in the row $\boldsymbol{a}$ and column $\boldsymbol{b}$ is equal to*

$$\mathsf{ACT}(\boldsymbol{a}, \boldsymbol{b}) = r_{\overline{b} \cdot S}(\overline{a})$$

Proposition 10 provides an equivalent description of an S-Box that has linear structure from the the autocorrelation of its component functions. Autocorrelation table can then be used to determine if an S-Box has linear structure.

Theorem 5. *An S-Box S has a linear structure if and only if there exists a nonzero $\overline{\alpha} \in \mathbb{F}_2^n$ and a nonzero $\overline{b} \in \mathbb{F}_2^m$ such that $\mathsf{ACT}(\boldsymbol{\alpha}, \boldsymbol{b}) = \pm 2^n$.*

Proof. This is an immediate consequences from Definition 3 and Proposition 10. □

Remark 2. Let $\overline{\alpha}$ be an input difference to S and let

$$\Omega_{\overline{\alpha}} = \{\overline{\beta} \in \mathbb{F}_2^m \mid \mathbf{Pr}_S[\overline{\alpha} \to \overline{\beta}] > 0\}$$

be the set of all possible output differences of S corresponding to $\overline{\alpha}$. If the entry $\mathrm{ACT}(\boldsymbol{\alpha}, \boldsymbol{b}) = +2^n$ (resp. -2^n), for $\overline{b} \in \mathbb{F}_2^m$, then $\overline{b} \cdot \overline{\beta} = 0$ (resp. 1) for all $\overline{\beta} \in \Omega_{\overline{\alpha}}$.

To determine if an S-Box has undisturbed bits, it is sufficient to observe nonzero row entries in each column of autocorrelation table that correspond to the autocorrelation spectrum of coordinate functions of the S-Box, i.e. the column $\mathbf{2}^i,\ i \in \{0, \dots, m-1\}$. The result is given as the following corollary.

Corollary 2. *For a nonzero input difference $\overline{\alpha}$, the i-th bit of the output difference of S is undisturbed if and only if $\mathsf{ACT}(\boldsymbol{\alpha}, \mathbf{2}^i) = \pm 2^n$, for $i \in \{0, \dots, m-1\}$.*

Proof. From Theorem 2, the vector $\overline{\alpha}$ is a linear structure of the coordinate function h_i. Clearly this is a direct consequence of Theorem 5. □

Autocorrelation table of the S-Box of Present is provided in Table 2. Some input differences that have undisturbed bits in its corresponding output differences can be observed in column **1**, which is the autocorrelation spectrum of the rightmost coordinate function. One may see in row entries **1**, **8**, and **9** at column **1** have value $\pm 2^4 = \pm 16$. Note that the row index represents the input difference and the column index represents the component functions of the S-Box. The magnitude of the entry indicate the value of undisturbed bits, where the sign "+" and "−" correspond to the undisturbed bit value equal to zero and one, respectively.

In Table 2 one may also find component functions, other than the coordinate functions, which have linear structures. For instance, the component functions in S-Box of Present represented by $\mathbf{10} \cdot S(\overline{x})$ and $\mathbf{11} \cdot S(\overline{x})$ have nontrivial linear structures (this can be seen in column **10** and **11** in Table 2 where some of the nonzero row entries are equal to $\pm 2^n$). The implication of this result was given in Remark 2. However, it remains unknown whether the existence of linear structures in component functions of an S-Box other than the coordinate functions could improve or lead to a new approach in (truncated)-differential cryptanalysis of bit-oriented block cipher.

5 S-Boxes with Undisturbed Bits

Recall from Theorem 2 that an S-Box has undisturbed bits if the derivative of any of its coordinate function at a nonzero vector in $\mathbb{F}_2^n$ is a constant function. The existence of an S-Box that has undisturbed bits can then be reduced into a question whether any of the coordinate functions of the S-Box has a nonzero linear structure.

So far the known Boolean functions that have nonzero linear structures are affine functions (from Proposition 8). If an S-Box has affine coordinate function, then definitely the S-Box has undisturbed bits. However, this is unlikely to occur in real case. This will lead to a linear approximation that involves input and output bits of the S-Box with probability one, and clearly does not serve its purpose as a nonlinear layer for block ciphers.

In order to find Boolean functions with linear structure, Proposition 6 restrict our attention to the Boolean functions of low degree. The following result is due to Carlet [4]. The complete proof of the following lemma is given in the appendix.

Table 2. Autocorrelation table of the S-Box of PRESENT. Column **1** correspond to the autocorrelation spectrum of the rightmost coordinate function h_0. Notice that the row entries **1**, **8**, **9** are equal to ± 16. Thus, for input difference **1**, **8**, **9**, the 0-th bit of the output difference of PRESENT's S-Box is undisturbed. The value of undisturbed bits is either 0 or 1, depending whether the magnitude is $+$ or $-$, respectively.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
0	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16	16
1	16	−16	0	0	0	0	0	0	0	0	−16	16	0	0	0	0
2	16	0	0	−8	−8	0	−8	8	0	−8	0	0	0	0	0	8
3	16	0	−8	0	0	−8	0	0	8	0	0	0	−8	−8	8	0
4	16	0	0	−8	−8	0	0	0	0	−8	0	0	−8	8	0	8
5	16	0	8	0	0	−8	−8	−8	−8	0	0	0	0	0	8	0
6	16	0	−8	8	0	0	0	0	−8	8	0	−16	0	0	0	0
7	16	0	0	0	0	0	8	−8	0	0	0	−16	8	−8	0	0
8	16	−16	−8	8	0	0	0	0	−8	8	0	0	0	0	0	0
9	16	16	0	0	−8	−8	0	0	0	0	0	0	0	0	−8	−8
10	16	0	0	−8	0	8	−8	8	0	−8	0	0	0	0	−8	0
11	16	0	8	0	8	0	0	0	−8	0	0	0	−8	−8	0	−8
12	16	0	0	−8	0	8	0	0	0	−8	0	0	−8	8	−8	0
13	16	0	−8	0	8	0	−8	−8	8	0	0	0	0	0	0	−8
14	16	0	0	0	0	0	0	0	0	0	−16	0	0	0	0	0
15	16	0	0	0	−8	−8	8	−8	0	0	16	0	8	−8	−8	−8

Lemma 3 [4]**.** *If f is a balanced n-variable Boolean function with* $\deg(f) = 2$, *then there exists a nonzero $\overline{\alpha} \in \mathbb{F}_2^n$ such that $D_{\overline{\alpha}} f(\overline{x}) = f(\overline{x}) \oplus f(\overline{x} \oplus \overline{\alpha}) = 1$ for all $\overline{x} \in \mathbb{F}_2^n$.*

We extend the result from Lemma 3 in Theorem 6 to show that an S-Box with at least one quadratic coordinate function has undisturbed bits. Hence we show that one may determine whether an S-Box has undisturbed bits from the degree of its coordinate functions.

Theorem 6. *Let S be a balanced $n \times m$ S-Box and $h_{m-1}, \ldots, h_0$ be its coordinate functions. If there exists a coordinate function h_i with* $\deg(h_i) = 2$ *then the S-Box S has undisturbed bits. More precisely, there exists a nonzero $\overline{\alpha} \in \mathbb{F}_2^n$ such that for input difference $\overline{\alpha}$, the i-th bit of the output difference of S is undisturbed and its value is* 1.

Proof. From Proposition 9, for every nonzero $\overline{b} \in \mathbb{F}_2^m$ all the component functions $\overline{b} \cdot S(\overline{x})$ are balanced Boolean functions, including the coordinate functions $h_{m-1}, \ldots, h_0$ of S. If there exists a coordinate function h_i with $\deg(h_i) = 2$, Lemma 3 says that there is a nonzero $\overline{\alpha} \in \mathbb{F}_2^n$ such that $D_{\overline{\alpha}} h_i(\overline{x}) = 1$ for all $\overline{x} \in \mathbb{F}_2^n$. Theorem 2 implies that for input difference $\overline{\alpha}$, the i-th bit of the output difference of S is undisturbed and its value is 1. □

Corollary 3. *If S is a balanced $n \times m$ S-Box with $n = 3$, then S has undisturbed bits. Moreover, for every $i \in \{0, \ldots, m-1\}$ there exists a nonzero $\overline{\alpha} \in \mathbb{F}_2^n$ such that for input difference $\overline{\alpha}$, the i-th bit of the output difference of S is undisturbed and its value is* 1.

Proof. Since S is a balanced S-Box, based on Proposition 1 then $\deg(\overline{b} \cdot S) \leq 2$ for all nonzero $\overline{b} \in \mathbb{F}_2^m$. It follows that every coordinate functions of S is of degree ≤ 2. The results follows immediately from Theorem 6 and Proposition 8. □

In [16] it was stated that every bijective 3×3 S-Box has undisturbed bits. Since bijective 3×3 S-Boxes are balanced S-Boxes, it follows immediately from Corollary 3 that they have undisturbed bits. This can be seen as an alternative proof of [16] where the author used the equivalence classes of 3×3 bijective S-Boxes.

Corollary 4. *Every* 3×3 *bijective S-Box has undisturbed bits.*

6 Conclusion and Further Remarks

In this work we define the notion of undisturbed bits of an S-Box and give its relation with other properties. S-Boxes which have undisturbed bits are shown to be a special class of S-Boxes with linear structures. We also show that it is possible to indicate whether an S-Box has undisturbed bits or not by using DDT and LAT. Autocorrelation table of an S-Box can be used as a dedicated tool to find nonzero input differences which have undisturbed bits in its output differences. The last result of this paper is the existence of undisturbed bits for balanced $n \times m$ S-Boxes with quadratic coordinate functions.

While the notion of undisturbed bits is related to the existence of nonzero linear structures in the coordinate functions of an S-Box, we also showed that other component functions of an S-Box may have nonzero linear structures. It remains unknown whether this property in an S-Box could improve or lead to a new approach in cryptanalysis of bit-oriented block ciphers.

7 Appendix

7.1 Proof of Lemma 3

Before proving the result in Lemma 3, the following two propositions are required.

Proposition 12 [4]**.** *Let f be n-variable Boolean function. We have the following relation*

$$\mathcal{W}_f^2(\overline{0}) = \sum_{\overline{b} \in \mathbb{F}_2^n} \mathcal{W}_{D_{\overline{b}} f}(\overline{0})$$

Proof.

$$\sum_{\overline{b}\in\mathbb{F}_2^n} \mathcal{W}_{D_{\overline{b}}f}(\overline{0}) = \sum_{\overline{b}\in\mathbb{F}_2^n}\left[\sum_{\overline{x}\in\mathbb{F}_2^n}(-1)^{D_{\overline{b}}f(\overline{x})}(-1)^{\overline{0}\cdot\overline{x}}\right] = \sum_{\overline{b}\in\mathbb{F}_2^n}\left[\sum_{\overline{x}\in\mathbb{F}_2^n}(-1)^{D_{\overline{b}}f(\overline{x})}\right]$$
$$= \sum_{\overline{b}\in\mathbb{F}_2^n} r_f(\overline{b}) = \sum_{\overline{b}\in\mathbb{F}_2^n} r_f(\overline{b})(-1)^{\overline{0}\cdot\overline{b}} = \mathcal{W}_f^2(\overline{0})$$

□

Proposition 13 [4]**.** *If f is an n-variables Boolean function with* $\deg(f) = 2$ *then*

$$\mathcal{W}_f^2(\overline{0}) = 2^n \sum_{\overline{b}\in\mathcal{LS}_f}(-1)^{D_{\overline{b}}f(\overline{0})}$$

Proof. Since the degree of f is equal to 2, it follows from Proposition 6 that for every $\overline{b}\in\mathbb{F}_2^n$ we have $\deg(D_{\overline{b}}f)\le 1$. Clearly $D_{\overline{b}}f$ is affine, hence from Proposition 2 it is either balanced (for nonzero coefficient vector) or constant function (for zero coefficient vector). Consequently, for the case where $D_{\overline{b}}f$ is balanced, we have $\mathcal{W}_{D_{\overline{b}}f}(\overline{0}) = 0$ from Proposition 3. Using the result from the Proposition 12, then

$$\mathcal{W}_f^2(\overline{0}) = \sum_{\overline{b}\in\mathbb{F}_2^n}\mathcal{W}_{D_{\overline{b}}f}(\overline{0}) = \sum_{\overline{b}\in\mathcal{LS}_f}\mathcal{W}_{D_{\overline{b}}f}(\overline{0}) = \sum_{\overline{b}\in\mathcal{LS}_f}\left[\sum_{\overline{x}\in\mathbb{F}_2^n}(-1)^{D_{\overline{b}}f(\overline{x})}\right]$$
$$= 2^n\sum_{\overline{b}\in\mathcal{LS}_f}(-1)^{D_{\overline{b}}f(\overline{0})}$$

□

Lemma 3 stated that if f is a balanced n-variable Boolean function with $\deg(f) = 2$, then there exist a nonzero $\overline{\alpha}\in\mathbb{F}_2^n$ such that $D_{\overline{\alpha}}f(\overline{x}) = f(\overline{x})\oplus f(\overline{x}\oplus\overline{\alpha}) = 1$ for all $\overline{x}\in\mathbb{F}_2^n$. The proof is given below.

Proof. Let f be a balanced n-variable Boolean function with $\deg(f) = 2$. Since f is balanced, then $\mathcal{W}_f(\overline{0}) = 0$ and consequently $\mathcal{W}_f^2(\overline{0}) = 0$. The result from Proposition 13 implies that the sum $\sum_{\overline{b}\in\mathcal{LS}_f}(-1)^{D_{\overline{b}}f(\overline{0})}$ must be equal to zero. We know that the zero vector $\overline{0}\in\mathbb{F}_2^n$ is a trivial linear structure because $D_{\overline{0}}f(\overline{x}) = 0$ for all $\overline{x}\in\mathbb{F}_2^n$. Clearly $\overline{0}\in\mathcal{LS}_f$. Using existence of zero vector in the set of linear structure of f, then there must exist a vector $\overline{\alpha}\in\mathbb{F}_2^n$, $\overline{\alpha}\ne\overline{0}$ such that $D_{\overline{\alpha}}f(\overline{x}) = 1$ for all $\overline{x}\in\mathbb{F}_2^n$. □

7.2 Linear Structures and Output Differences of an S-Box

Theorem 7. *Let S be an $n\times m$ S-Box and $\Omega_{\overline{\alpha}} = \{\overline{\beta} = (\beta_{m-1},\ldots,\beta_0)\in\mathbb{F}_2^m \mid$ $\mathbf{Pr}_S[\overline{\alpha}\to\overline{\beta}] > 0\}$ be the set of all possible output differences of S corresponding to input difference $\overline{\alpha}\in\mathbb{F}_2^n$. The vector $\overline{\alpha}$ is a linear structure of the component function $\overline{b}\cdot S(\overline{x})$ if and only if $\overline{b}\cdot\overline{\beta}$ remains equal for all $\overline{\beta}\in\Omega_{\overline{\alpha}}$.*

Proof. Let $h_{m-1}, \ldots, h_0$ be coordinate functions of the S-Box S. For the vector $\overline{b} = (b_{m-1}, \ldots, b_0) \in \mathbb{F}_2^m$ we can express the component function $\overline{b} \cdot S(\overline{x})$ as a linear combination of coordinate functions of S, i.e. $\overline{b} \cdot S(\overline{x}) = b_{m-1}h_{m-1}(\overline{x}) \oplus \ldots \oplus b_0h_0(\overline{x})$. Since $\overline{\alpha} \in \mathbb{F}_2^n$ is a linear structure of $\overline{b} \cdot S(\overline{x})$, we have the following

$$
\begin{aligned}
c &= \overline{b} \cdot S(\overline{x}) \oplus \overline{b} \cdot S(\overline{x} \oplus \overline{\alpha}) && \forall \overline{x} \in \mathbb{F}_2^n \\
c &= (b_{m-1}h_{m-1}(\overline{x}) \oplus \ldots \oplus b_0h_0(\overline{x})) \oplus \\
&\qquad (b_{m-1}h_{m-1}(\overline{x} \oplus \overline{\alpha}) \oplus \ldots \oplus b_0h_0(\overline{x} \oplus \overline{\alpha})) && \forall \overline{x} \in \mathbb{F}_2^n \\
c &= b_{m-1}(h_{m-1}(\overline{x}) \oplus h_{m-1}(\overline{x} \oplus \overline{\alpha})) \oplus \ldots \oplus b_0(h_0(\overline{x}) \oplus h_0(\overline{x} \oplus \overline{\alpha})) && \forall \overline{x} \in \mathbb{F}_2^n \\
c &= \overline{b} \cdot (h_{m-1}(\overline{x}) \oplus h_{m-1}(\overline{x} \oplus \overline{\alpha}), \ldots, h_0(\overline{x}) \oplus h_0(\overline{x} \oplus \overline{\alpha})) && \forall \overline{x} \in \mathbb{F}_2^n \\
c &= \overline{b} \cdot \overline{\beta} && \forall \overline{\beta} \in \Omega_{\overline{\alpha}}
\end{aligned}
$$

The converse is obvious from above equations. □

7.3 DDT of the S-Box of PRESENT

See Table 3.

Table 3. Difference distribution table of the S-Box of PRESENT.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
0	16	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	4	0	0	0	4	0	4	0	0	0	4	0	0
2	0	0	0	2	0	4	2	0	0	0	2	0	2	2	2	0
3	0	2	0	2	2	0	4	2	0	0	2	2	0	0	0	0
4	0	0	0	0	0	4	2	2	0	2	2	0	2	0	2	0
5	0	2	0	0	2	0	0	0	0	2	2	2	4	2	0	0
6	0	0	2	0	0	0	2	0	2	0	0	4	2	0	0	4
7	0	4	2	0	0	0	2	0	2	0	0	0	2	0	0	4
8	0	0	0	2	0	0	0	2	0	2	0	4	0	2	0	4
9	0	0	2	0	4	0	2	0	2	0	0	0	2	0	4	0
10	0	0	2	2	0	4	0	0	2	0	2	0	0	2	2	0
11	0	2	0	0	2	0	0	0	4	2	2	2	0	2	0	0
12	0	0	2	0	0	4	0	2	2	2	2	0	0	0	2	0
13	0	2	4	2	2	0	0	2	0	0	2	2	0	0	0	0
14	0	0	2	2	0	0	2	2	2	2	0	0	2	2	0	0
15	0	4	0	0	4	0	0	0	0	0	0	0	0	0	4	4

References

1. Biham, E., Shamir, A.: Differential cryptanalysis of DES-like cryptosystems. J. Cryptol. **4**(1), 3–72 (1991)
2. 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)
3. Carlet, C.: Vectorial Boolean functions for cryptography. In: Crama, Y., Hammer, P.L. (eds.) Boolean Models and Methods in Mathematics, Computer Science, and Engineering, pp. 398–469. Cambridge University Press, Cambridge (2010)
4. Carlet, C.: Boolean functions for cryptography and error correcting codes. In: Crama, Y., Hammer, P.L. (eds.) Boolean Models and Methods in Mathematics, Computer Science, and Engineering, pp. 257–397. Cambridge University Press, Cambridge (2010)
5. 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)
6. Evertse, J.-H.: Linear structures in block ciphers. In: Price, W.L., Chaum, D. (eds.) EUROCRYPT 1987. LNCS, vol. 304, pp. 249–266. Springer, Heidelberg (1988)
7. Knudsen, L.R.: Truncated and higher order differentials. In: Preneel [13], pp. 196–211
8. Lai, X.: Additive and linear structures of cryptographic functions. In: Preneel [13], pp. 75–85
9. Lai, X., Maurer, U.: Higher order derivatives and differential cryptanalysis. In: Blahut, R., Costello, D.J., Maurer, U., Mittelholzer, T. (eds.) Communications and Cryptography. The Springer International Series in Engineering and Computer Science, vol. 276, pp. 227–233. Springer, New York (1994)
10. Matsui, M.: Linear cryptanalysis method for DES cipher. In: Helleseth, T. (ed.) EUROCRYPT 1993. LNCS, vol. 765, pp. 386–397. Springer, Heidelberg (1994)
11. Meier, W., Staffelbach, O.: Nonlinearity criteria for cryptographic functions. In: Quisquater, J.-J., Vandewalle, J. (eds.) EUROCRYPT 1989. LNCS, vol. 434, pp. 549–562. Springer, Heidelberg (1990)
12. Preneel, B.: Analysis and Design of cryptographic hash functions. Ph.D. thesis, Katholieke Universiteit Leuven (1993), rené Govaerts and Joos Vandewalle (promotors)
13. Preneel, B. (ed.): FSE 1994. LNCS, vol. 1008. Springer, Heidelberg (1995)
14. Sarkar, P., Maitra, S.: Construction of nonlinear Boolean functions with important cryptographic properties. In: Preneel, B. (ed.) EUROCRYPT 2000. LNCS, vol. 1807, pp. 485–506. Springer, Heidelberg (2000)
15. Sun, S., Hu, L., Wang, P.: Automatic security evaluation for bit-oriented block ciphers in related-key model: application to PRESENT-80, LBlock, and others. IACR Cryptology ePrint Archive 2013, 676 (2013)
16. Tezcan, C.: Improbable differential attacks on PRESENT using undisturbed bits. J. Comput. Appl. Math. **259**(Part B(0)), 503–511 (2014)
17. Zhang, W., Bao, Z., Lin, D., Rijmen, V., Yang, B., Verbauwhede, I.: RECTANGLE: a bit-slice ultra-lightweight block cipher suitable for multiple platforms. IACR Cryptology ePrint Archive 2014, 84 (2014)
18. Zhang, X.M., Zheng, Y., Imai, H.: Relating differential distribution tables to other properties of substitution boxes. Des. Codes Cryptogr. **19**(1), 45–63 (2000)

Differential Sieving for 2-Step Matching Meet-in-the-Middle Attack with Application to LBlock

Riham AlTawy and Amr M. Youssef(✉)

Concordia Institute for Information Systems Engineering,
Concordia University, Montréal, QC, Canada
youssef@ciise.concordia.ca

Abstract. In this paper, we propose a modified approach for the basic meet-in-the-middle attack which we call differential sieving for 2-step matching. This technique improves the scope of the basic meet in the middle attack by providing means to extend the matching point for an extra round through differential matching and hence the overall number of the attacked rounds is extended. Our approach starts by first reducing the candidate matching space through differential matching, then the remaining candidates are further filtered by examining non shared key bits for partial state matching. This 2-step matching reduces the total matching probability and accordingly the number of remaining candidate keys that need to be retested is minimized. We apply our technique to the light weight block cipher LBlock and present a two known plaintexts attack on the fifteen round reduced cipher. Moreover, we combine our technique with short restricted bicliques and present a chosen plaintext attack on Lblock reduced to eighteen rounds.

Keywords: Cryptanalysis · Meet-in-the-middle · Differential sieving · LBlock · Short bicliques

1 Introduction

Meet in the middle (MitM) attacks have drawn a lot of attention since the inception of the original attack which was first proposed in 1977 by Diffie and Hellman [13] for the analysis of the Data Encryption Standard (DES). Soon after, the attack became a generic approach to be used for the analysis of ciphers with non complicated key schedules. For this class of ciphers, one can separate the execution into two independent parts where each part can be computed without guessing all the bits of the master key. The first execution part covers encryption rounds from the plaintext to some intermediate state and the other part covers decryption rounds from the corresponding ciphertext to the same internal state. At this point, the attacker has knowledge of the same intermediate state from two independent executions where the right key guess produces matching states. A typical MitM attack can be launched with as low as one known

© Springer International Publishing Switzerland 2015
T. Eisenbarth and E. Öztürk (Eds.): LightSec 2014, LNCS 8898, pp. 126–139, 2015.
DOI: 10.1007/978-3-319-16363-5_8

plaintext-ciphertext pair. Accordingly, with the recent growing interest in low data complexity attacks [9], the MitM attack has witnessed various improvements and has been widely adopted for the analysis of various cryptographic primitives. The increasing motivation for adopting low data complexity attacks for the analysis of ciphers is backed by the fact that security bounds are better perceived in a realistic model. Particularly, in a real life scenario, security protocols impose restrictions on the amount of plaintext-ciphertext pairs that can be eavesdropped and/or the number of queries permitted under the same key.

With the current popularity of lightweight devices such as RFID chips and wireless sensor networks, the demand for efficient lightweight cryptography is increasing. These devices offer convenient services on tiny resource constrained environments with acceptable security and privacy guarantees. In particular, the employed ciphers must obey the aggressive restrictions of the application environment while maintaining acceptable security margins. As a result, the designers of lightweight ciphers are often forced to make compromising decisions to fulfil the required physical and economical constraints. Among the designs that have been proposed to address these needs are PRESENT [7], KATAN and KTANTAN [12], LED [15], Zorro [14], and LBlock [25]. All of these proposals have received their fair share of cryptanalytic attacks targeting their weak properties [4,8,18,19,21]. Such unfavourable properties usually are the result of the desire of the designers to conform to the resources constraints.

The success of the MitM attack depends of the speed of key diffusion. Complex key schedules amount for quick diffusion and hence the knowledge of the state after few rounds involves all the bits in the master key. Indeed, one can say that the witnessed renewal of interest in MitM attacks is due to the emergence of lightweight ciphers which often tend to employ simple key schedules with relatively slow diffusion to meet the resources constraints.

In this work, we present *differential sieving for 2-step matching*, a technique that improves the scope and the key retesting phase of the basic meet in the middle attack. More precisely, our technique enables the attacker to cover at least one extra round in an execution direction when all state knowledge is not available. This extension is accomplished by matching possible differential transitions through the Sbox layer instead of matching actual state values. Afterwards, the remaining candidate keys from both directions are used to evaluate the state for only one round, which is further matched by the actual bit values. The proposed 2-step differential-value matching reduces the total matching probability and accordingly the number of remaining key candidates that need to be retested is minimized. We demonstrate our technique on the light weight block cipher LBlock and present a two known plaintexts attack on the fifteen round reduced cipher. Finally, we combine our approach with restricted short bicliques and present an eighteen round attack with a data complexity of 2^{17}.

The rest of the paper is organized as follows. In the next section, we explain our proposed technique and give a brief overview on the basic meet-in-the-middle attack and the idea of short restricted bicliques. Afterwards, in Sect. 3, we give the specification of the lightweight block cipher Lblock and provide detailed

application of our attack on it. Specifically, we present a low-data complexity attack on the fifteen round reduced cipher and a restricted biclique attack on the cipher reduced to eighteen rounds. Finally, the paper is concluded in Sect. 4.

2 Differential Sieving for 2-Step Matching

Our approach is a modified approach of the basic meet-in-the-middle attack [13] which was first proposed in 1977. Throughout the following years, the attack has been used in the security analysis of a large number of primitives including block ciphers, stream ciphers, and hash functions. The basic attack has undergone major improvements to make it better suit the attacked primitive. In particular, the cut and splice technique [2] and the initial structure approach [20] are successfully used in MitM preimage attacks on hash function [1,24]. Moreover, partial matching [3] allows the matching point to cover more rounds through matching only known parts of the state. Other examples of these techniques include 3-subset MitM [8] and sieve-in-the-middle [10]. It is worth noting that most MitM attacks are low data complexity attacks except when used with bicliques [6], which represent a more formal description of the initial structure and can be constructed from related key differentials.

2.1 Basic Meet-in-the-middle Attack

The basic MitM attack recovers the master key of a given cipher more efficiently than by brute forcing it. As depicted in Fig. 1, the attack idea can be explained as follows: let the attacked primitive be an r-round block cipher operating under a fixed master key K. Let $E_{i,j}(p)$ denote the partial encryption of the plaintext p and $D_{i,j}(c)$ denote the partial decryption of the ciphertext c, where i and j are the starting and ending rounds for the partial encryption/decryption, respectively. If one can compute $E_{1,j}(p)$ using K_1 and $D_{r,j}(c)$ using K_2 such that K_1 and K_2 do not share some key bits and each key guess does not involve the whole master key, then the same state of the cipher can be computed independently from the encryption and decryption sides. More precisely, each guess of K_1 allows us to compute a candidate $E_{1,j}(p)$ and each guess of K_2 gives a candidate $D_{r,j}(c)$. Since the output of both executions is state j, then all the guessed (K_1, K_2) pairs that result in $E_{1,j}(p) = D_{r,j}(c)$ are considered potentially right keys and one of them must be the right key.

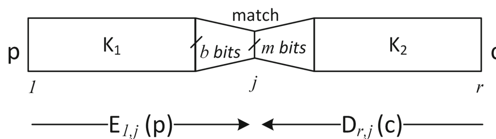

Fig. 1. Basic MitM attack

Generally, given one plaintext-ciphertext pair, the number of potentially right keys depends on the number of bits that are matched at state j. More formally, given a b-bit state, a k-bit master key, and the knowledge of m bits at the matching state, the number of potentially right keys is 2^{k-m}, where 2^{-m} is the probability that the two states match at the available m bits. Partial matching takes place if $m < b$ and hence the 2^{k-m} candidate keys need to be retested for full state matching. In the case of $m = b$, only the relation between the key size k and the state size b determines the number of potentially right keys. If $b = k$, then only the right key remains and no further testing is required. However, in some cipher, designers tend to use master keys that are larger than the state size in order to provide a higher security margin in more constrained environments. Accordingly, even if we are performing full state matching, we end up with 2^{k-b} potentially right keys. Consequently, more plaintext-ciphertext pairs are needed to find the right key. Indeed, whether we are partially or fully matching the states, if $k > b$, we get a set of 2^{k-b} potentially right keys after retesting with one plaintext-ciphertext pair. In this case, to recover the right key, the data complexity of the MitM attack is $n = \lceil \frac{k}{b} \rceil$ plaintext-ciphertext pairs.

2.2 Differential Sieving Approach

The time complexity of the MitM attack is divided into two main components: the MitM part, where both forward and backward computations take place, and the key retesting part where the remaining potentially right keys need to be rechecked. More formally, let $K_s = K_1 \cap K_2$ be the set of bits shared between K_1 and K_2, $K_f = K_1 \backslash K_1 \cap K_2$ be the set of bits in K_1 only, and $K_b = K_2 \backslash K_1 \cap K_2$ be the set of bits in K_2 only. We also assume that the master key $K = K_1 \cup K_2$. The time complexity of the MitM attack is given by:

$$\underbrace{2^{|K_s|}(2^{|K_f|} \cdot c_f + 2^{|K_b|} \cdot c_b)}_{\text{MitM}} + \underbrace{2^{|K|} \times 2^{-\rho_s} \cdot c_t}_{\text{Key testing}},$$

where $2^{-\rho_s}$ is the total matching probability, c_f and c_b denotes the costs of partial computations in the forward and backward directions, and $c_t = c_f + c_b$ is the total cost of computation. According to this equation, the MitM attack recovers the right key more efficiently than exhaustively searching the master key space if both K_f and K_b are non empty sets and the number of remaining keys that need retesting is not too large. In other words, besides efficient separation of executions, the key retesting part of the MitM attack should not be the component dominating the attack time complexity. Evidently, the lower the total matching probability $2^{-\rho_s}$ is, the less keys remain for rechecking.

Our proposed *differential sieving 2-step matching* approach provides the means to decrease the total matching probability by matching both the possible differential transitions through the Sbox layer and actual partial state bit values. The technique also allows us to extend the basic MitM by at least one more round when the key knowledge is not available anymore. Indeed, when we have knowledge of parts of the state before the subkey mixing, we lose this

state knowledge if the bits of this subkey are not in the guessed key set. However, if we are using two plaintext-ciphertext pairs, we know the difference after the key mixing with certainty. To this end, we can extend our state difference knowledge with one extra round and use it for matching possible input/output differences through the following substitution layer. Our technique requires at least two plaintext-ciphertext pairs which is fortunate for two reasons: first, we match two different states variables thus we get lower total matching probability. Second, when we analyze a light weight cipher whose key size is larger than the block size (the case for LBlock), the MitM attack requires more than a single plaintext-ciphertext pair to recover the right key anyway.

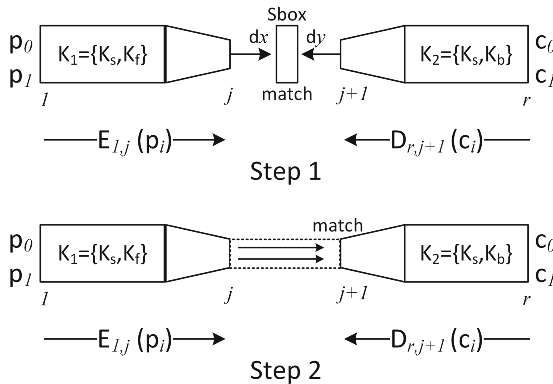

Fig. 2. Difference-value 2-step matching MitM attack

Figure 2 depicts the high level idea of the proposed 2-step matching approach and shows how it differs from the basic approach shown in Fig. 1. The core idea of the attack is to efficiently separate the execution such that we get the same partial state knowledge (value or difference) from both the backward and forward directions. Let Xf_j^i and Xb_j^i be the states at round j resulting from the forward and backward executions using the ith plaintext and ciphertext, respectively. As depicted in Fig. 2, the forward execution ends at round j where we are forced to terminate the execution because the available knowledge about the m-bit partial state Xf_j^i will be lost after the key mixing in round $j+1$. More precisely, let K_m be the set of bits in the master key that are Xored with the m-bit partial state Xf_j^i at round $j+1$. If $K_m \notin K_f$ then we lose the matching knowledge in state Xf_{j+1}^i. However, since the master key $K = K_1 \cup K_2$, then with certainty $K_m \in K_b$. The procedure of our attacks is described as follows:

- For each guess k_s of the $2^{|K_s|}$ values of K_s
  - For each guess k_f of the $2^{|K_f|}$ values, partially encrypt the two plaintexts p_0 and p_1, and store the resulting m-bit partial states values (Xf_j^0, Xf_j^1) and difference $dx = (Xf_j^0 \oplus Xf_j^1)$ in a table T

- For each guess k_b of the $2^{|K_b|}$ values, partially decrypt the two corresponding ciphertexts c_0 and c_1 to get the two m-bit partial states values (Xb^0_{j+1}, Xb^1_{j+1}).
  1. Step 1: check if the resulting difference $dy = (Xb^0_{j+1} \oplus Xb^1_{j+1})$ and any of the dx entries in T is a possible differential transition through the Sbox layer. If yes then go to Step 2, else guess another k_b.
  2. Step 2: get K_m from k_b and using the forward table T entry which matched in step 1, compute $Xf^0_{j+1} = Xf^0_j \oplus K_m$ and $Xf^1_{j+1} = Xf^1_j \oplus K_m$. Check if these two states are equal to Xb^0_{j+1}, Xb^1_{j+1}. If yes then add the current master key candidate $(k_s \cup k_f \cup k_b)$ to a list L_p of potentially right keys, else guess another k_b.
- Exhaustively test if any of the candidate keys in L_p encrypts both plaintexts p_0 and p_1 to their corresponding ciphertexts c_0 and c_1. If yes output it as the correct master key, else guess another k_s

Other than extending the matching point by one more round, the main gain of the 2-step matching is reducing the total matching probability and hence the false positive keys whose retesting may dominate the total attack complexity. More formally, given an m-bit matching knowledge, let the probability of a possible differential for a given Sbox be $2^{-\alpha}$, then the total matching probability in our approach is $2^{-\rho_s} = 2^{-\alpha} \times 2^{-m} \times 2^{-m}$. Assuming $c_f \approx c_b \approx c_t/2$, the total time complexity of the attack is given by:

$$2^{|K_s|}(2^{|K_f|} + 2^{|K_b|}) \cdot c_t + 2^{|K|-\alpha-2m} \cdot c_t$$

The memory complexity is given by $2^{|K_f|} + 2^{|K_f|+|K_b|-\alpha-2m}$. This complexity may be negligible as the sizes of both lists T and L_p are usually small. The data complexity is equal to the unicity distance of the analyzed cipher.

2.3 Short Restricted Bicliques

Biclique cryptanalysis [6] was first used to present an accelerated exhaustive search on the full round AES. The basic idea of bicliques is to increase the number of rounds of the basic MitM attack. As depicted in Fig. 3, a biclique is a structure of the 3-tuple $\{p^u_j, s^u_i, K_q\}$ where K_q denotes the key bits used to encrypt the plaintext p to an intermediate state s. K_q is partitioned into three disjoint sets of key bits $K_q = \{K_5, K_3, K_4\}$ such that for a given u of the $2^{|K_5|}$ values of K_5, the states variables between p and s that are affected by a change in the value of K_3 are different than those affected by a change in the value of K_4. More formally, let $Enc_{[u,i,j]}(p)$ and $Dec_{[u,i,j]}(s)$ denote the encryption and decryption of the plaintext p and intermediate state s using the u, i, and j values of K_5, K_3, and K_4, respectively. A biclique can be constructed if for all u, i, and j of the $2^{|K_5|}, 2^{|K_3|}$, and $2^{|K_4|}$ values, respectively, the computation of $s^u_i = Enc_{[u,i,0]}(p)$ does not share any active state variables with the computation $p^u_j = Dec_{[u,0,j]}(s)$. Since both the forward and backward computations generate two independent differential paths, one can deduce that $s^u_i = Enc_{[u,i,j]}(p^u_j)$.

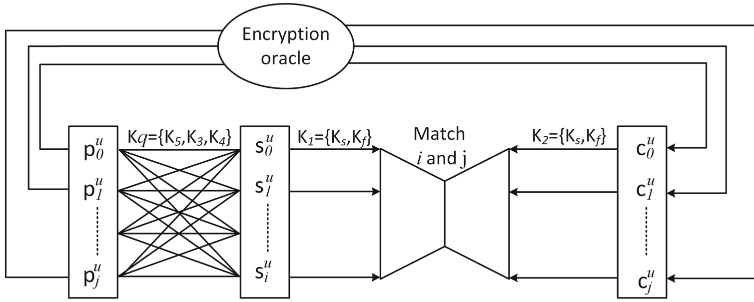

Fig. 3. Combining short restricted bicliques with basic MitM attack

In our attack we employ short restricted bicliques [10]. They are restricted in the sense that we do not have the freedom in partitioning K_q as in the case of conventional biclique cryptanalysis [6]. More precisely, we first separate the forward and backward execution of the MitM attack to get K_1 and K_2. In the sequel, the choice of K_3, K_4, and K_5 must conform to some restrictions. Particularly, $K_3 \in K_f$, $K_4 \in K_b$, and $K_5 \in K_s$ so that each choice of K_s, K_f, and K_b of the 2-step matching MitM attack gives a unique $K_q = \{K_5, K_3, K_4\}$ candidate that can be used to construct the biclique. For a given u of the $2^{|K_5|}$ values, a biclique is constructed as follows:

- Get the base states (p_0^u, s_0^u): choose the base plaintext $p_0^u = 0$ and set $K_q = \{K_5, K_3, K_4\} = \{u, 0, 0\}$. The base intermediate state $s_0^u = Enc_{[u,0,0]}(p_0^u)$. Note that $p_0^u = 0$ for all values of K_5 which is not the case for s_0^u as it depends on the value of K_5 used in K_q to partially encrypt p_0^u.
- Compute (p_j^u, s_i^u): for each i of the $2^{|K_3|}$ values of K_3, set $s_i^u = Enc_{[u,i,0]}(p_0^u)$ and for each j of the $2^{|K_4|}$ values of K_4, set $p_j^u = Dec_{[u,0,j]}(s_0^u)$.

To this end, we get $s_i^u = Enc_{[u,i,j]}(p_j^u)$ and an exhaustive search on K_5 is performed to construct the bicliques with time complexity of $2^{|K_5|}(2^{|K_3|} + 2^{|K_4|})$. Following [10], we include the biclique construction in the 2-step MitM and hence its time complexity is incorporated with the overall complexity of the MitM attack. Since we are using differential sieving, we need to build two base bicliques with two different base plaintext ${p_0}_0^u = 0$, and ${p_1}_0^u = 1$. In what follows we explain how we combine short restricted bicliques with differential sieving 2-step MitM attack as depicted in Fig. 4:

- For each guess k_s of the $2^{|K_s|}$ values, we get a guess u of the $2^{|K_5|}$ values of K_5
  - For each guess k_f of the $2^{|K_f|}$ values of K_f, we get a guess i of the $2^{|K_3|}$ values of K_3:
    1. Partially encrypt the two base plaintexts to get the corresponding intermediate states of the biclique ${s_0}_i^u = Enc_{[u,i,0]}({p_0}_0^u)$ and ${s_1}_i^u = Enc_{[u,i,0]}({p_1}_0^u)$. If $i = 0$, then save the base intermediate states ${s_0}_0^u$ and ${s_1}_0^u$

2. Partially encrypt the two intermediate states ${s_0}_i^u$ and ${s_1}_i^u$, and store the resulting m-bit partial states values (Xf_z^0, Xf_z^1) and difference $dx = (Xf_z^0 \oplus Xf_z^1)$ in a table T

– For each guess k_b of the $2^{|K_b|}$ values of K_b, we get a guess j of the $2^{|K_4|}$ values of K_4:
  1. Partially decrypt the two base intermediate states to get the corresponding plaintexts of the biclique ${p_0}_j^u = Dec_{[u,0,j]}({s_0}_0^u)$ and ${p_1}_j^u = Dec_{[u,0,j]}({s_1}_0^u)$.
  2. Ask the encryption oracle for the ciphertexts ${c_0}_j^u$ and ${c_1}_j^u$ corresponding to the plaintexts acquired in the previous step.
  3. Partially decrypt the two ciphertexts ${c_0}_j^u$ and ${c_1}_j^u$ to get the two m-bit partial states values (Xb_{z+1}^0, Xb_{z+1}^1).
  4. Perform the 2-step matching procedure described in the previous section.

– Exhaustively test if any of the candidate keys in L_p encrypts any two plaintexts ${p_0}_j^u$ and ${p_1}_j^u$ to their corresponding ciphertexts ${c_0}_j^u$ and ${c_1}_j^u$. If yes output it as the correct master key, else guess another k_s

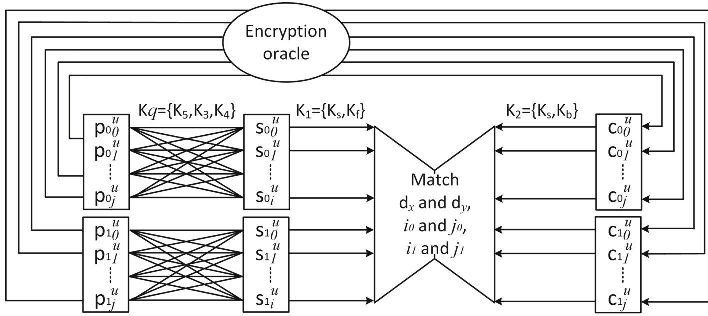

Fig. 4. Combining short restricted bicliques with differential sieving 2-step matching MitM attack

The total time complexity is composed of three parts: biclique construction, MitM, and key retesting. More formally, let c_q be the cost of biclique computation, the total time complexity of the attack is given by:

$$2 \times 2^{|K_5|}(2^{|K_3|} + 2^{|K_4|}) \cdot c_q + 2^{|K_s|}(2^{|K_f|} + 2^{|K_b|}) \cdot c_t + 2^{|K|-\alpha-2m} \cdot c_t.$$

The memory complexity is given by $2^{|K_f|} + 2^{|K_f|+|K_b|-\alpha-2m} + 2^{(\#\text{ of active bits in } p)+1}$ where the third term denotes the memory needed to store the chosen plaintext-ciphertext pairs for all the j and u values of K_3 and K_5 for two bicliques. Regardless of the value of K_3 and K_5, we only get a small number of plaintexts since they always differ in only few places. The data complexity is $2^{(\#\text{ of active bits in } p)+1}$ chosen plaintexts to get the corresponding ciphertexts of the plaintexts produced by the two bicliques.

3 Application to LBlock

In this section we demonstrate our technique on the lightweight block cipher LBlock [25]. LBlock requires about 1320 GE on 0.18 μm technology, which satisfies the required limitation of 2000 GE in RFID applications. LBlock has been analyzed with respect to various types of attacks [5,22] including: impossible differential [16,17], integral [21], boomerang [11], and biclique [23] cryptanalysis. Note that the attack presented in [23] is a typical high data complexity biclique cryptanalysis where the whole key space is exhaustively searched. More precisely, similar to the biclique attack on AES [6], in [23], the authors presented an attack with a time complexity $\approx 2^{79}$, almost equal to that of the average exhaustive search. The factor of 2 gain is due to counting the number of Sboxes which are affected by the difference only and not evaluating the whole state. In addition, the attack has a data complexity of 2^{52}.

3.1 Notation

Besides the notations used in describing our technique in Sect. 2, the notation used in applying our attack on LBlock is as follows:

- K: The master key.
- Sk_i: i^{th} round sub key.
- X_i: The eight 4-bit nibble state at round i.
- $X_i[j]$: j^{th} nibble of the ith round state.
- $K_{[i,j]}$: i^{th} and j^{th} bits of master key K.

3.2 Specifications of LBlock

LBlock [25] is a 64-bit lightweight cipher with an 80-bit master key. It employs a 32-round Feistel structure variant. Its internal state is composed of eight 4-bit nibbles. As depicted in Fig. 5, the round function adopts three nibble oriented transformations: subkey mixing, 4-bit Sboxes, and nibble permutation. The 80-bit master key K is stored in a key register denoted by $k = k_{79}k_{78}k_{77}.......k_1k_0$. The leftmost 32 bits of the register k are used as i^{th} round subkey Sk_i. The key register is updated after the extraction of each Sk_i as follows:

1. $k \lll 29$.
2. $[k_{79}k_{78}k_{77}k_{76}] = S_9[k_{79}k_{78}k_{77}k_{76}]$.
3. $[k_{75}k_{74}k_{73}k_{72}] = S_8[k_{75}k_{74}k_{73}k_{72}]$.
4. $[k_{50}k_{49}k_{48}k_{47}k_{46}] \oplus [i]_2$,

where S_8 and S_9 are two 4-bit Sboxes. For further details, the reader is referred to [25].

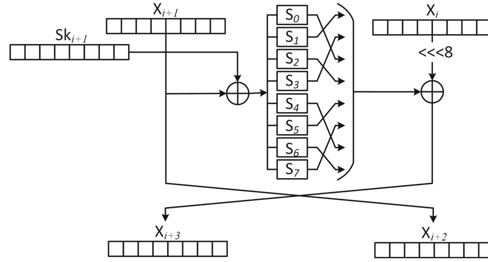

Fig. 5. LBock round function

3.3 Low Data-Complexity Attack on LBlock

In this section, we present the first low data complexity attack on reduced round LBlock. The attack exploits the weak diffusion of the key schedule. This fact enables us to find some nibbles in states at higher rounds whose values do not involve all the bits from the master key. Using symbolic evaluation of the cipher, we have found an execution separation for the first fifteen rounds of LBlock. The knowledge of the first nibble of the 8th round state requires guessing 73 and 72 bits of the master key when evaluating it from the forward and backward executions, respectively. More precisely, given a known plaintext-cipher text pair, the forward execution requires guessing $K - K_{[0,1,14,15,16,17,35]}$ to compute $Xf_8[0]$ and the backward execution requires guessing $K - K_{[55,62,63,64,65,66,67,68]}$ to compute $Xb_8[0]$. For this separation, we can use a basic MitM attack since we have the knowledge of one nibble from both directions at the same round. To recover the right key, we have to try $\lceil \frac{80}{64} \rceil = 2$ plaintext-ciphertext pairs. Accordingly, the total matching probability $2^{-4\times 2}$. Following the notation used in Sect. 2, $|K_s| = 65$, $|K_f| = 8$, and $|K_b| = 7$. Hence, the time complexity of this 2 known plaintext attack is given by $2^{65}(2^8 + 2^7) + 2^{80-8} = 2^{73}$ and a memory complexity of 2^8.

While the previous attack was an application of the basic MitM attack, Fig. 6 shows an execution separation for fifteen rounds of Lblock starting from round four. We opted for demonstrating the 2-step matching approach on these specific rounds because in the following section, we will add a short restricted biclique in the first three rounds. Our attack requires the knowledge of nibbles $Xf_{11}[1]$ and $Xf_{10}[6]$ in the forward computation and nibble $Xb_{12}[0]$ in the backward computation. The adopted execution separation is depicted in Fig. 6, where the red and blue colours denote known state nibbles in consecutive rounds in the forward and backward executions, respectively. The yellow colour denotes key nibbles that are guessed in each direction and the white colour represents unknown nibbles. We have found that the knowledge of $Xf_{11}[1]$ and $Xf_{10}[6]$ involves 76 bits from the master key, namely $K_1 = K - K_{[2,1,0,79]}$. In the backward direction, $Xb_{12}[0]$ requires guessing 73 bits, specifically $K_2 = K - K_{[26,27,27,29,30,31,32]}$. Accordingly, $|K_s| = 69$, $|K_f| = 7$, and $|K_b| = 4$.

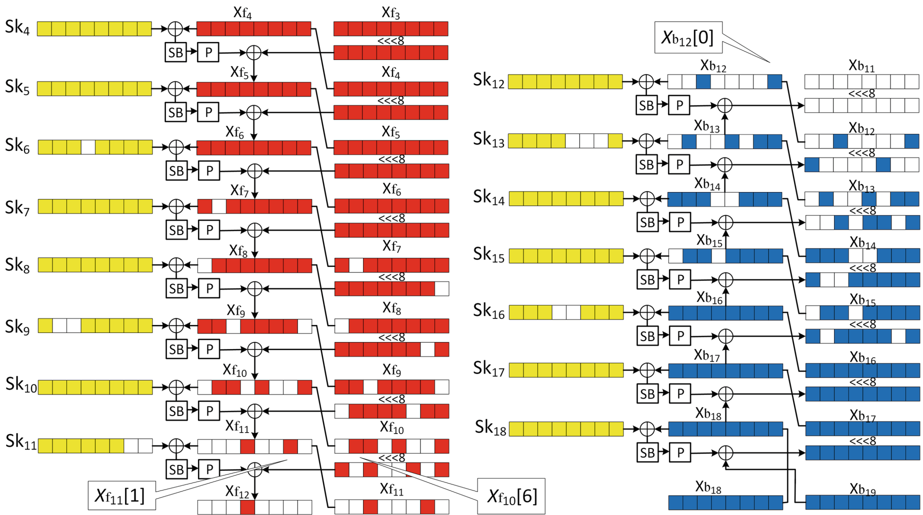

Fig. 6. Fifteen rounds execution separation for our 2-step matching MitM attack (Color figure online)

We assume that states Xf_3 and Xf_4 are loaded with the known plaintexts, while states Xb_{18} and Xb_{19} are loaded with their corresponding ciphertexts. The attack follows exactly the procedure defined in Sect. 2.2. However, due to the fact that LBlock employs a Feistel structure, the matching process involves two different states from the forward direction. Particularly, for the two known plaintexts p_0 and p_1, we store in table T nibbles $(Xf^0_{11}[1], Xf^1_{11}[1])$ and $(Xf^0_{10}[6], Xf^1_{10}[6])$ along with the guessed K_1. For each guess of K_2 in the backward execution using the two corresponding ciphertexts c_0 and c_1, we compute $Xb^0_{12}[0]$ and $Xb^1_{12}[0]$, and we apply the 2-step matching as follows:

1. Set $dx = Xf^0_{11}[1] \oplus Xf^1_{11}[1]$ and $dy = P^{-1}((Xb^0_{12}[0] \oplus Xb^1_{12}[0]) \oplus ((Xf^0_{10}[6] \oplus Xf^1_{10}[6])))$, where P^{-1} is the inverse permutation of the round function. Check if (dx, dy) is a possible differential pair from the differential distribution table (DDT) of Sbox S_1.
2. From the current K_b knowledge, one can get a candidate value for $Sk_{11}[1]$ which is unknown in the forward direction and consequently a candidate $(Xf^0_{12}[0], Xf^1_{12}[0])$. Match these two candidate nibbles with $Xb^0_{12}[0]$ and $Xb^1_{12}[0]$.

If a (K_1, K_2) pair passed the 2-step matching then it is considered for retesting. The probability of possible differentials of the Sbox S_1 is $\approx 2^{-1.4}$, and we match two different nibbles, hence the total matching probability is $2^{-1.4-4-4}$. This two known plaintext-ciphertext attack has a time complexity of $2^{69}(2^7 + 2^4) + 2^{80-9.4} \approx 2^{76}$, and a memory complexity of 2^7.

3.4 Three More Rounds with Restricted Bicliques

The previous fifteen round attack is extended to eighteen rounds by appending three round restricted biclique. As depicted in Fig. 7, if $K_3 = K_{[27,28,29,30]}$ and $K_4 = K_{[0,1,79]}$, the differential paths in the first three rounds are independent. More precisely, a change in K_3 affects $Sk_2[2]$ which consecutively propagates the effect to the red nibbles in the forward direction. In the backward direction, changing K_4 involves $Sk_1[7]$ and $Sk_3[2]$, their effect is denoted by the blue nibbles in the states. Since $K_3 \neq K_f$ and $K_4 \neq K_b$, we have to consider bits $K_{[2,26,31,32]}$ as shared bits, i.e., $K_s = K_s + K_{[2,26,31,32]}$ and thus $|K_5| = 73$. The attack follows the procedure presented in Sect. 2.3, and the 2-step matching is performed exactly as in the fifteen round attack. Since $c_q \approx c_t/4$, following the time complexity equation in Sect. 2.3, this chosen plaintext attack has a time complexity of $2^{73-1}(2^4+2^3)+2^{73}(2^4+2^3)+2^{80-9.4} \approx 2^{78}$. The memory and data complexities are determined by the number of active bits in the plaintexts of the bicliques. Since we have four active nibbles in the plaintexts of both bicliques, this attack has a memory and data complexities of 2^{17}.

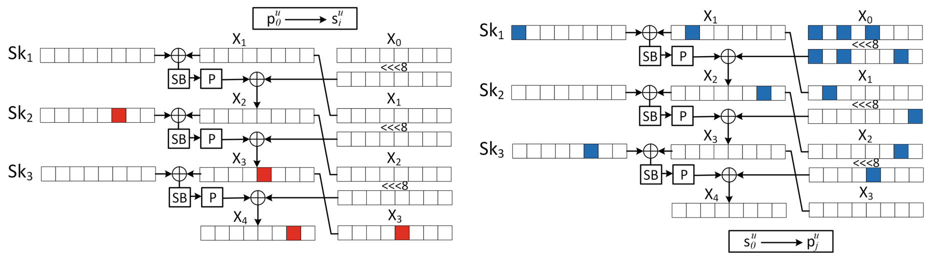

Fig. 7. First three rounds restricted biclique (Color figure online)

4 Conclusion

We have presented a modified approach to decrease the total matching probability of the MitM attack. This approach extends the attack by one more round and incorporates differential and value matching to reduce the number of false positive keys. We have shown how to combine our approach with short restricted bicliques and demonstrated the approach on the light weight block cipher LBlock. Particularly, we have presented a two known plaintexts 2-step matching MitM attack on fifteen rounds with time complexity of 2^{76} and memory complexity of 2^7. Finally, we have combined the fifteen rounds execution with 3-round restricted bicliques and an eighteen round chosen plaintext attack is presented with time, memory, and data complexities of 2^{78}, 2^{17}, and 2^{17}, respectively.

Acknowledgment. The authors would like to thank the anonymous reviewers for their valuable comments and suggestions that helped improve the quality of the paper.

This work is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) and Le Fonds de Recherche du Québec - Nature et Technologies (FRQNT).

References

1. AlTawy, R., Youssef, A.M.: Preimage attacks on reduced-round stribog. In: Pointcheval, D., Vergnaud, D. (eds.) AFRICACRYPT. LNCS, vol. 8469, pp. 109–125. Springer, Heidelberg (2014)
2. 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)
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. Bar-On, A., Dinur, I., Dunkelman, O., Lallemand, V., Tsaban, B.: Improved analysis of zorro-like ciphers. Cryptology ePrint Archive, Report 2014/228 (2014)
5. Bogdanov, A., Boura, C., Rijmen, V., Wang, M., Wen, L., Zhao, J.: Key difference invariant bias in block ciphers. In: Sako, K., Sarkar, P. (eds.) ASIACRYPT 2013, Part I. LNCS, vol. 8269, pp. 357–376. Springer, Heidelberg (2013)
6. Bogdanov, A., Khovratovich, D., Rechberger, C.: Biclique cryptanalysis of the full AES. In: Lee, D.H., Wang, X. (eds.) ASIACRYPT 2011. LNCS, vol. 7073, pp. 344–371. Springer, Heidelberg (2011)
7. 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)
8. 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)
9. Bouillaguet, C., Derbez, P., Dunkelman, O., Fouque, P.-A., Keller, N., Rijmen, V.: Low-data complexity attacks on AES. IEEE Trans. Inf. Theory **58**(11), 7002–7017 (2012)
10. Canteaut, A., Naya-Plasencia, M., Vayssière, B.: Sieve-in-the-middle: improved MITM attacks. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013, Part I. LNCS, vol. 8042, pp. 222–240. Springer, Heidelberg (2013)
11. Chen, J., Miyaji, A.: Differential cryptanalysis and boomerang cryptanalysis of LBlock. In: Cuzzocrea, A., Kittl, C., Simos, D.E., Weippl, E., Xu, L. (eds.) CD-ARES Workshops 2013. LNCS, vol. 8128, pp. 1–15. Springer, Heidelberg (2013)
12. 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)
13. Diffie, W., Hellman, M.: Exhaustive cryptanalysis of the NBS data encryption standard. Computer **10**(6), 74–84 (1977)
14. Gérard, B., Grosso, V., Naya-Plasencia, M., Standaert, F.-X.: Block ciphers that are easier to mask: how far can we go? In: Bertoni, G., Coron, J.-S. (eds.) CHES 2013. LNCS, vol. 8086, pp. 383–399. Springer, Heidelberg (2013)
15. 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)

16. Karakoç, F., Demirci, H., Harmancı, A.E.: Impossible differential cryptanalysis of reduced-round LBlock. In: Askoxylakis, I., Pöhls, H.C., Posegga, J. (eds.) WISTP 2012. LNCS, vol. 7322, pp. 179–188. Springer, Heidelberg (2012)
17. 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)
18. Rijmen, V., Toz, D., Mendel, F., Varici, K.: Differential analysis of the LED block cipher. In: Wang, X., Sako, K. (eds.) ASIACRYPT 2012. LNCS, vol. 7658, pp. 190–207. Springer, Heidelberg (2012)
19. Nakahara Jr., J., Sepehrdad, P., Zhang, B., Wang, M.: Linear (Hull) and algebraic cryptanalysis of the block cipher PRESENT. In: Garay, J.A., Miyaji, A., Otsuka, A. (eds.) CANS 2009. LNCS, vol. 5888, pp. 58–75. Springer, Heidelberg (2009)
20. Sasaki, Y.: Meet-in-the-middle preimage attacks on AES hashing modes and an application to Whirlpool. In: Joux, A. (ed.) FSE 2011. LNCS, vol. 6733, pp. 378–396. Springer, Heidelberg (2011)
21. Sasaki, Y., Wang, L.: Meet-in-the-middle technique for integral attacks against feistel ciphers. In: Knudsen, L.R., Wu, H. (eds.) SAC 2012. LNCS, vol. 7707, pp. 234–251. Springer, Heidelberg (2013)
22. Wang, Y., Wu, W.: Improved multidimensional zero-correlation linear cryptanalysis and applications to LBlock and TWINE. In: Susilo, W., Mu, Y. (eds.) ACISP 2014. LNCS, vol. 8544, pp. 1–16. Springer, Heidelberg (2014)
23. Wang, Y., Wu, W., Yu, X., Zhang, L.: Security on LBlock against biclique cryptanalysis. In: Lee, D.H., Yung, M. (eds.) WISA 2012. LNCS, vol. 7690, pp. 1–14. Springer, Heidelberg (2012)
24. Wu, S., Feng, D., Wu, W., Guo, J., Dong, L., Zou, J.: (Pseudo) preimage attack on round-reduced Grøstl hash function and others. In: Canteaut, A. (ed.) FSE 2012. LNCS, vol. 7549, pp. 127–145. Springer, Heidelberg (2012)
25. 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)

Match Box Meet-in-the-Middle Attacks on the SIMON Family of Block Ciphers

Ling Song[1,2,3](✉), Lei Hu[1,2], Bingke Ma[1,2,3], and Danping Shi[1,2,3]

[1] State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China
[2] Data Assurance and Communication Security Research Center, Chinese Academy of Sciences, Beijing 100093, China
[3] University of Chinese Academy of Sciences, Beijing 100049, China
{lsong,hu,bkma,dpsh}@is.ac.cn

Abstract. SIMON is a family of lightweight block ciphers designed by the U.S National Security Agency in 2013. In this paper, we analyze the resistance of the SIMON family of block ciphers against the recent match box meet-in-the-middle attack which was proposed in FSE 2014. Our attack particularly exploits the weaknesses of the linear key schedules of SIMON. Since the data available to the adversary is rather limited in many concrete applications, it is worthwhile to assess the security of SIMON against such low-data attacks.

Keywords: Lightweight block cipher · SIMON · Meet-in-the-middle attack · Match box

1 Introduction

Recent years have witnessed the rapid development of lightweight cryptography. In order to meet the uprising security demands in resource restrained environment such as RFID tags and wireless sensor networks, many lightweight block ciphers have been proposed as the fundamental cryptographic primitives, including PRESENT [6], KATAN & KTANTAN [9], PRINTcipher [14], LBlock [19], Piccolo [16], LED [11], SIMON & SPECK [3], to name but a few. In order to design a cipher satisfying resource constraints, the inner components should be simple and easy to implement, especially the key schedules. For example, KATAN, PRINCE and SIMON have linear key schedules and LED uses the master key directly without any key schedule.

These new design styles give rise to new cryptanalytic techniques. In particular, meet-in-the-middle attacks have developed a lot in the analysis of lightweight block ciphers [7,8]. New techniques, such as indirect matching, all-subkey recovery [17], bicliques [13], sieve-in-the-middle [8] and match box [10] have been proposed and make meet-in-the-middle attacks more powerful in cryptanalysis.

Meet-in-the-middle attacks normally can apply to ciphers with simple key schedules. As a unique instance, the recently proposed match box technique [10] aims particularly at block ciphers with linear key schedules.

© Springer International Publishing Switzerland 2015
T. Eisenbarth and E. Öztürk (Eds.): LightSec 2014, LNCS 8898, pp. 140–151, 2015.
DOI: 10.1007/978-3-319-16363-5_9

SIMON is a family of lightweight block ciphers designed by the U.S National Security Agency in 2013. It attracts many researchers since there is no internal security analysis of the cipher included in the specification document. In the literature, analyses of SIMON focus on differential cryptanalysis [4] and linear cryptanalysis [15]. Typical examples are [1,2,5,18] which belong to the sort of statistical cryptanalysis and thus require a great data complexity. However, in this paper we only concentrate on attacks with low data complexity.

Our contributions. In this paper, we analyze the resistance of the SIMON family of block ciphers against the match box meet-in-the-middle attack. Compared with classical statistical attacks such as differential and linear attacks [4,15], the match box meet-in-the-middle attack on SIMON requires much less and more reasonable amount of data. To the best of our knowledge, this is the first meet-in-the-middle type of attack on SIMON, which is both meaningful and attractive for many concrete protocols and applications, where only a small amount of plaintext/ciphertext pairs can be eavesdropped by the adversary. Our work enriches the analytical results on SIMON in the literature.

The remainder of this paper is organized as follows. The notations used in this paper are defined in Sect. 2; we recall the match box meet-in-the-middle attack in Sect. 3; Sect. 4 briefly describes the family of block ciphers SIMON, elaborates the match box meet-in-the-middle attack against the smallest version of SIMON and summarizes the results of other SIMON versions; we conclude the paper in the last section.

2 Notations

The following notations will be used throughout this paper:

P, C	: plaintext and ciphertext.
$E_{0-R}(K, P)$	: encryption of P from round 0 to round R with the secret key K.
$D_{R-R_1}(K, C)$	: decryption of C from round R to round R_1 with the secret key K.
X^i	: the i-th state word.
X^i_j	: the j-th bit of the word X^i.
$\|K\|$	: the bit size of K or the dimension of the linear space generated by K.

3 Match Box Meet-in-the-Middle Attack

In this section, we recall the details of the match box meet-in-the-middle attack [10] and some related techniques.

3.1 Basic Meet-in-the-Middle Attack

As depicted in Fig. 1, the basic meet-in-the-middle attack assumes that a fraction of the internal state v could be computed from a plaintext P with a portion K_1 of the master key K, and that v could also be computed from the corresponding ciphertext C with a portion K_2 of K.

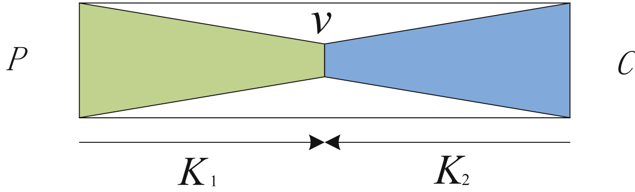

Fig. 1. Basic meet-in-the-middle attack

Assume $K_1 \cup K_2 = K$, $K_1 \cap K_2 = K_\cap$, $K'_1 = K_1 - K_\cap$ and $K'_2 = K_2 - K_\cap$. The basic meet-in-the-middle attack proceeds in two stages: a key filtering stage which sieves out the key candidates, followed by a verification stage that tests each candidate to derive the right key. The first stage with one pair of plaintex/ciphertext is as follows:

- For each $k_\cap \in K_\cap$:
  1. For each $k'_1 \in K'_1$, v is computed, and store the corresponding pair (v, k'_1) in a table indexed by v.
  2. For each $k'_2 \in K'_2$, v is computed. From the previous table, retrieve the k'_1s by matching v. The combination of k'_1, k'_2 and $k_\cap$ forms a candidate master key.

The right key is necessarily among the candidate keys, since it must lead to a match at any internal state. After the key filtering stage described above, the key space is reduced to $2^{|K|-|v|}$.

In the verification stage, each candidate key is tested. To find the right key, $N = \lceil \frac{|K|}{n} \rceil$ pairs of plaintext/ciphertext are needed, where n is the block size.

In total, the time complexity is:

$$
\begin{aligned}
T &= T_{\text{Filtering}} + T_{\text{Verifying}} \\
&= 2^{K_\cap} \cdot \left(2^{|K'_1|} \cdot \frac{R_1}{R} + 2^{|K'_2|} \cdot \frac{R_2}{R} \right) + \sum_{i=0}^{N-1} 2^{|K|-|v|-i*n}
\end{aligned}
$$

encryptions, where R_1 and R_2 are the number of rounds in forward computation and backward computation respectively, and $R = R_1 + R_2$ is the number of rounds attacked. The memory complexity is $\min\{2^{|K'_1|}, 2^{|K'_2|}\}$, and the data complexity is N known plaintext/ciphertext pairs.

A simple tweak can be utilized to reduce the complexity. Suppose t pairs of plaintext/ciphertext are used in the first stage. Then the complexity of the first stage rises t times, while in the second stage only $2^{|K|-t\cdot|v|}$ candidate keys need to be tested.

The indirect matching technique, which neglects the round key bits which have a linear impact on the matching point, can also be exploited to decrease the complexity. Suppose $v = E_{0-R_1}(K_1, P) = D_{R-R_1}(K_2, C)$ and $L(K_1), L(K_2)$

are the bits of K_1, K_2 that have a linear impact on v. Then this equation can be rewritten as

$$v = E_{0-R_1}(K_1 - L(K_1), P) \oplus L(K_1) = D_{R-R_1}(K_2 - L(K_2), C) \oplus L(K_2),$$

which is equivalent to

$$E_{0-R_1}(K_1 - L(K_1), P) \oplus D_{R-R_1}(K_2 - L(K_2), C) = L(K_1) \oplus L(K_2). \quad (1)$$

A correct guess of $K_1 - L(K_1), K_2 - L(K_2)$ makes Eq. (1) hold for different pairs of plaintext/ciphertext, say (P_1, C_1) and (P_2, C_2), that is,

$$\begin{aligned} &E_{0-R_1}(K_1 - L(K_1), P_1) \oplus D_{R-R_1}(K_2 - L(K_2), C_1) \\ =&E_{0-R_1}(K_1 - L(K_1), P_2) \oplus D_{R-R_1}(K_2 - L(K_2), C_2). \end{aligned}$$

In this way, if multiple pairs of plaintext/ciphertext are used, key bits in $L(K_1)$, $L(K_2)$ can be excluded and thus less bits need to be considered from both directions of computation in the first stage. Consequently, the complexity decreases accordingly.

3.2 Match Box Meet-in-the-Middle Attack

The match box technique was proposed in [10] which fits in the general sieve-in-the-middle framework [8]. As shown in Fig. 2, l is computed from a plaintext P with $k_1 \in K_1$ and r is computed from the corresponding ciphertext C with $k_2 \in K_2$. The match box is a precomputed table that stores all compatible (l, r)s under control of $k_3 \in K_3$, which has a small size. The simplest case is that $K_3 \cap K_1 = K_3 \cap K_2 = \varnothing$. In this case, once l and r are computed, the match box returns whether l and r are compatible. If so, the corresponding (k_1, k_2, k_3) is a candidate key. In more practical cases, K_3 may involve bits from both K_1 and K_2, which makes it difficult to construct a match box with a reasonable complexity.

In [10], the authors proposed a method to construct match boxes for block ciphers with linear key schedules. Following the previous notations, $K_1 \cap K_2 = K_\cap, K_1' = K_1 - K_\cap$ and $K_1 \cup K_2 = K$. K_3 is related to both K_1 and K_2. Let f be the key schedule function. Since $K = K_1' + K_2$, then $f(K) = f(K_1') \oplus f(K_2)$. On the right side of the match box, r and k_2 are known (which means $f(K_2)$ is also known) and considered as a whole $\vec{r} = (r, f(K_2))$. On the left side of the match box, (l, k_1') are known. l depends on k_1' and there are $2^{l \cdot |K_1'|}$ mappings of $g : k_1' \mapsto l$.

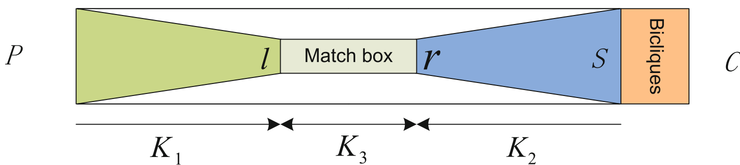

Fig. 2. Match box meet-in-the-middle attack

Given a mapping g, for each value of $\vec{r}$ we can precompute a list of k_1's (at most $2^{|K_1'|}$) leading to l such that (l, k_1') and $\vec{r}$ are compatible. There are $2^{|\vec{r}|}$ values of $\vec{r}$, and as a result $2^{|K_1'|+|\vec{r}|}$ computations should be stored for each mapping g. Enumerating all possible mappings, the match box is supposed to have

$$2^{|l|\cdot 2^{|K_1'|}+|K_1'|+|\vec{r}|} \tag{2}$$

entries in total[1].

During the attack, l is computed using k_1 in the forward computation and the mapping g is thus determined; in the backward computation, $\vec{r}$ is computed. To verify the compatibility between (l, k_1') and $\vec{r}$, we search for the list of k_1's using $\vec{r}$ among the items of the match box which is indexed by the mapping g, and check whether the k_1' used in forward computation is in that list.

The main limitation of this technique is the size of the match box table. More precisely, as specified in Eq. (2), the table becomes too large if $|K_1'|$ is not small enough.

On the key separations. Following the idea in [10], the master key can be regarded as a vector in $(\mathbb{Z}/2\mathbb{Z})^{|K|}$. The value of the master key corresponds to the coordinates of this vector along the canonical basis. Each round key is a linear combination of the master key bits, and then $|K_1|$ (resp. $|K_2|$) can be regarded as the dimension of the linear space generated by K_1 (resp. K_2). In this way, it becomes much easier to find independent separations of the round key bits, which results in more efficient meet-in-the-middle attacks for KATAN in [10]. This strategy is very likely to be applicable to other block ciphers with linear key schedules.

4 Match Box Meet-in-the-Middle Attack on SIMON

4.1 The SIMON Family of Block Ciphers

SIMON is a family of lightweight block ciphers designed by NSA which aims to provide an optimal hardware implementation performance [3]. SIMON supports a variety of word sizes $n = 16, 24, 32, 48$ and 64 bits. SIMON$2n$ with m n-bit key words is denoted by SIMON$2n/mn$. Table 1 makes explicit the parameter choices for all versions of SIMON.

The design of SIMON follows a classical Feistel structure, operating on two n-bit halves in each round. The round function makes use of three n-bit operations: XOR ($\oplus$), AND ($\&$) and circular shift ($\lll$). Given a round key k it is defined on two inputs x and y as

$$R_k(x, y) = (y \oplus f(x) \oplus k, x),$$

where $f(x) = ((x \lll 1)\&(x \lll 8)) \oplus (x \lll 2)$. In this paper, (X^0, X^1) denotes the plaintext and (X^i, X^{i+1}) denotes the state after i rounds of encryption.

[1] We have confirmed from the authors of [10] that the complexity is not $2^{|l|^{|K_1'|}+|K_1'|+|\vec{r}|}$ as their paper describes, but $2^{|l|\cdot 2^{|K_1'|}+|K_1'|+|\vec{r}|}$.

Table 1. SIMON parameters.

Block size $2n$	Key size mn	Word size n	Key words m	Rounds T
32	64	16	4	32
48	72	24	3	36
48	96	24	4	36
64	96	32	3	42
64	128	32	4	44
96	96	48	2	52
96	144	48	3	54
128	128	64	2	68
128	192	64	3	69
128	256	64	4	72

The key schedules of SIMON are linear transformations, as depicted in Fig. 4. The m master key words are used as the first m round keys. Also, they are the inputs for the first iteration of the key schedules. Note that, the i-th round key is denoted as K^i. The key schedule differs slightly for different m. The constant c is $2^n - 4$ and z_js are 1-bit constant sequences. For more specifications of SIMON, please refer to [3] Fig. 3.

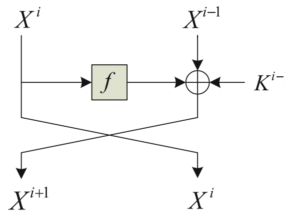

Fig. 3. The round function of SIMON

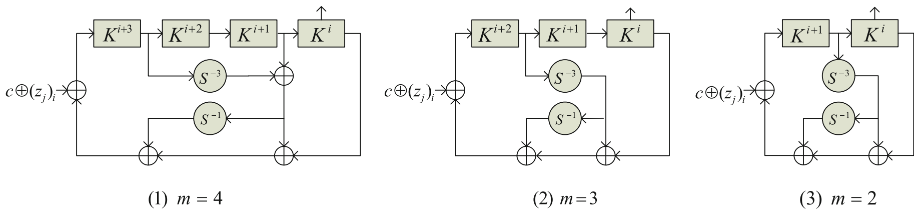

Fig. 4. The key schedules for $m = 4, 3, 2$. S^j operation denotes left circular shift by j bits.

4.2 Application of Match Box Meet-in-the-Middle Attack to SIMON32/64

SIMON32/64 is the version of SIMON with 16-bit words and 64-bit keys ($n = 16, m = 4$). As depicted in Fig. 5, we need to guarantee that $|K_1| < 64$ in the forward computation, such that the attack is faster than the brute-force attack. Following the algorithm in [10] that determines key dependencies by marking the key bits that enter the state bits and propagating the dependencies along the cipher, we also write a program to observe the dependency between internal state bits and key bits. According to our computation, after seven rounds of encryption each state bit are influenced by 63 round key bits, among which 61 bits have a nonlinear impact on the state bit. These 61 round key bits form a linear space of dimension 61.

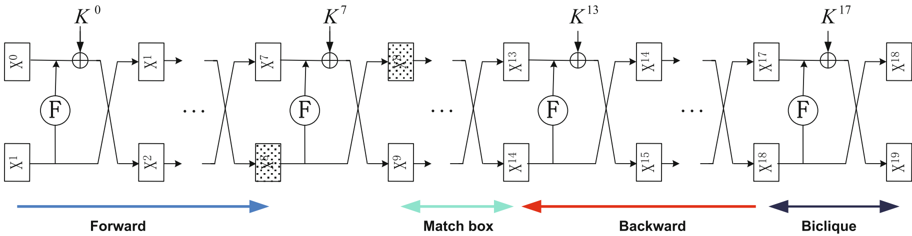

Fig. 5. Match box meet-in-the-middle attack on SIMON32/64

Similarly in the backward computation, 61 round key bits have a nonlinear impact on each state bit after seven rounds of decryption, which also form a linear space of dimension 61.

Setting any bit of state X^8 as the matching point, we are able to mount a basic meet-in-the-middle attack on 15-round SIMON32/64 using 3 plaintext/ciphertext pairs. The complexity is

$$3 \cdot \left(2^{61} \cdot \frac{7}{15} + 2^{61} \cdot \frac{7}{15}\right) + \sum_{i=0}^{2} 2^{61-i*32} = 2^{62.93}.$$

Extend the Basic Attack to More Rounds. Suppose the matching point is X_0^8, i.e. the least significant bit of X^8. According to the round function,

$$X_0^8 = X_{15}^9 \cdot X_8^9 \oplus X_{14}^9 \oplus X_0^{10} \oplus K_0^8. \tag{3}$$

By decomposing X_{15}^9 in Eq. (3), we get

$$X_0^8 = (X_{14}^{10} \cdot X_7^{10} \oplus X_{13}^{10} \oplus X_{15}^{11} \oplus K_{15}^9) \cdot X_8^9 \oplus X_{14}^9 \oplus X_0^{10} \oplus K_0^8.$$

Here we neglect the round key bits which have a linear impact on X_0^8, so K_0^8 is neglected. Besides K_0^8, several other round key bits may be neglected as

long as they impact on X_0^8 linearly. From the decryption side, to get the value of X_0^8, we just need to calculate the values of $X_{15}^{11}, X_0^{10}, X_7^{10}, X_{13}^{10}, X_{14}^{10}, X_8^9$ and X_{14}^9. Thus, the round key bit K_{15}^9 is isolated from the backward computation, and will be processed in the match box. As can be seen, more round key bits of K_2 will be isolated from the backward computation if the decomposition goes further, leading a reduction on the dimension of K_2. To keep $K_2 = 61$, more round key bits after the 15-th round can be added. In this way the number of rounds attacked may increase.

From the above analysis, we can derive the following principle: we can move the backward computation beyond the 15-th round by isolating more round key bits in the middle; however we need to keep the dimension of K_2 to be 61. In other words, the load of the backward computation keeps the same, but the overall number of rounds attacked may increase since more middle rounds can be covered with the match box.

In fact, the number of round key bits that can be isolated is determined by the construction of the match box. The round key bits in the middle involve information form both K_1 and K_2, and they can not be computed independently in any direction. Since SIMON adopts linear key schedules and $K = K_1' + K_2$, each round key bit can be split into two parts: $K_j^i = rk_j^i \oplus lk_j^i$, where rk_j^i denotes the part generated by K_2 and lk_j^i denotes the part generated by K_1'.

In this case, $|K_1'| = 3$ and $|l| = 3$ if three plaintext/ciphertext pairs are used; $\vec{r}$ consists of the state bits and rk_j^is, which is absorbed from the decomposed expression of X_0^8. According to Eq. (2), the match box requires a memory complexity of $M = 2^{27+|\vec{r}|}$.

Compression table. Normally, $|\vec{r}|$ should be smaller than 37 to guarantee $M < 2^{64}$. In order to cover as many middle rounds with the match box as possible, we utilize the so-called *compression table* technique of [8] to shorten $\vec{r}$ when $|K_1'|$ is small. This technique comes from the fact that the decomposed expression of X_0^8 can be expressed as a boolean polynomial in the bits of K_1'. It is obvious that the boolean polynomial in the bits of K_1' has no more than $2^{|K_1'|}$ coefficients, to which all bits in $\vec{r}$ can be mapped. Therefore, if $2^{|K_1'|} < 2^{|\vec{r}|}$, $|\vec{r}|$ bits from the right side of the match box can be equivalently expressed as at most $2^{|K_1'|}$ coefficients. Consequently, the memory complexity of the match box gets reduced.

However, the cost of building a compression table that transforms bits of $\vec{r}$ into coefficients of bits in K_1' cannot be neglected. Since the coefficients can be computed with $2^{|K_1'|}$ partial encryptions for each $\vec{r}$, the whole compression table is built with time complexity $2^{|\vec{r}|+|K_1'|}$ and memory complexity $2^{|\vec{r}|}$.

The decomposition shows X_8^0 can be represented with 27 state bits and 32 rk_j^is, and the attack can now be extended to 17 rounds. We have $|K_1'| = 3, |\vec{r}| = 27 + 32 = 55 > 2^3$ and a compression table of size 2^{55} will be built with time complexity of 2^{58}.

If we match three plaintext/ciphertext pairs simultaneously, the three corresponding 55-bit $\vec{r}$ are shorten to $8 \times 3 = 24$ bits using the compression table. This yields a match box of size $2^{27+24} = 2^{51}$.

Total complexity. We attack 17 rounds of SIMON32/64, and the overall time complexity is

$$T_{Total} = T_{Compression} + T_{MatchBox} + T_{Filtering} + T_{Verifying}$$
$$= 2^{58} + 2^{51} + 3 \cdot (2^{61} \cdot \frac{7}{17} + 2^{61} \cdot \frac{4}{17}) + \sum_{i=0}^{3-1} 2^{61-32*i} \approx 2^{62.57}.$$

The round key bits in K_1, K_2 and $\vec{r}$ are provided in the Appendix. For all versions of SIMON, since the last round key is added to the state linearly, which means no overlapped nonlinear component, we can extend one round with bicliques [13] at the end of the cipher with no additional time complexity but 2^N chosen plaintext/ciphertext pairs, where N is 3 or 4. Finally, 18 rounds of SIMON32/64 can be attacked.

4.3 Application to Other Versions

For other versions of SIMON, similar attacks can be mounted. Table 2 summarizes the results of eight versions. We omit the results of the other two versions for $m = 2$, i.e. SIMON96/96 and SIMON128/128, because the match box meet-in-the-middle attack does not outperform the basic meet-in-the-middle attack which can attack 17 and 19 rounds of SIMON96/96 and SIMON128/128 respectively. The reason why the match box meet-in-the-middle attack does not work well for these two versions is due to the fact that any match box for them covering one round in the middle becomes too large. Note that, for the cases $m = 2$, the key size is as large as the block size, which makes it difficult to construct a match box with a reasonable complexity. For larger m, which means the key size is larger than block size, the match box meet-in-the-middle attack has been proved to be more efficient.

Table 2. Results for eight versions of SIMON

Version: $2n/mn$	Rounds: Total	Rounds: Attacked	Data	Time	$\|K_1\|$	$\|K_2\|$	Match box	Compression table
32/64	32	18	2^3	$2^{62.57}$	61	61	2^{51}	2^{55}
48/72	36	17	2^3	$2^{71.65}$	70	71	2^{18}	2^{47}
48/96	36	19	2^3	$2^{95.26}$	94	94	2^{18}	2^{72}
64/96	42	17	2^3	$2^{94.05}$	93	93	2^{35}	2^{65}
64/128	44	19	2^3	$2^{126.01}$	125	125	2^{35}	2^{76}
96/144	54	21	2^4	$2^{141.27}$	140	140	2^{132}	2^{96}
128/192	69	25	2^3	$2^{190.60}$	189	190	2^{51}	2^{97}
128/256	72	25	2^3	$2^{253.94}$	253	253	2^{35}	2^{116}

5 Conclusion

In this paper, we analyzed eight versions of the SIMON family of block ciphers with the match box meet-in-the-middle attack. These eight versions share a common feature that the key size is larger than the block size. Our work exploits the weaknesses of the linear key schedules of SIMON. Compared to the existing attacks based on statistical methods, our attack requires much less data, which is meaningful for many concrete situations where the data available to the adversary is rather limited.

Acknowledgement. The authors would like to thank anonymous reviewers for their helpful comments and suggestions. The work of this paper was supported by the National Key Basic Research Program of China (2013CB834203), the National Natural Science Foundation of China (Grants 61070172), the Strategic Priority Research Program of Chinese Academy of Sciences under Grant XDA06010702, and the State Key Laboratory of Information Security, Chinese Academy of Sciences.

A Details for the Attack on SIMON32/64

- K_1 involves 61 round key bits (dimension 61) as follows:
  $K_0^0, K_1^0, K_2^0, K_3^0, K_4^0, K_5^0, K_6^0, K_7^0, K_8^0, K_9^0, K_{10}^0, K_{11}^0, K_{12}^0, K_{13}^0, K_{14}^0, K_{15}^0,$
  $K_0^1, K_1^1, K_2^1, K_3^1, K_4^1, K_5^1, K_6^1, K_7^1, K_8^1, K_9^1, K_{10}^1, K_{11}^1, K_{12}^1, K_{13}^1, K_{14}^1, K_{15}^1,$
  $K_0^2, K_2^2, K_3^2, K_4^2, K_5^2, K_6^2, K_7^2, K_9^2, K_{10}^2, K_{11}^2, K_{12}^2, K_{13}^2, K_{14}^2,$
  $K_4^3, K_5^3, K_6^3, K_8^3, K_{11}^3, K_{12}^3, K_{13}^3, K_{14}^3, K_{15}^3,$
  $K_0^4, K_6^4, K_7^4, K_{13}^4, K_{14}^4,$
  $K_8^5, K_{15}^5.$
- The match box involves 29 round keys generated by $K2$:
  $rk_8^9, rk_{15}^9, rk_0^{10}, rk_6^{10}, rk_7^{10}, rk_{13}^{10}, rk_{14}^{10}, rk_4^{11}, rk_5^{11}, rk_6^{11}, rk_8^{11}, rk_{11}^{11}, rk_{12}^{11},$
  $rk_{13}^{11}, rk_{14}^{11}, rk_{15}^{11}, rk_0^{12}, rk_5^{12}, rk_6^{12}, rk_7^{12}, rk_{11}^{12}, rk_{12}^{12}, rk_{13}^{12}, rk_{14}^{12}, rk_8^{13}, rk_{13}^{13},$
  $rk_{15}^{13}, rk_{14}^{13}, rk_0^{14}.$
- K_2 involves 67 round key bits (dimension 61) as follows:
  $K_2^{12}, K_3^{12}, K_4^{12}, K_6^{12}, K_7^{12}, K_{10}^{12},$
  $K_0^{13}, K_1^{13}, K_2^{13}, K_3^{13}, K_4^{13}, K_5^{13}, K_6^{13}, K_7^{13}, K_8^{13}, K_9^{13}, K_{10}^{13}, K_{11}^{13}, K_{12}^{13},$
  $K_0^{14}, K_1^{14}, K_2^{14}, K_3^{14}, K_4^{14}, K_5^{14}, K_6^{14}, K_7^{14}, K_8^{14}, K_9^{14}, K_{10}^{14}, K_{11}^{14}, K_{12}^{14}, K_{13}^{14},$
  $K_{14}^{14}, K_{15}^{14},$
  $K_0^{15}, K_1^{15}, K_2^{15}, K_3^{15}, K_4^{15}, K_5^{15}, K_6^{15}, K_7^{15}, K_8^{15}, K_9^{15}, K_{10}^{15}, K_{11}^{15}, K_{12}^{15}, K_{13}^{15},$
  $K_{14}^{15}, K_{15}^{15},$
  $K_0^{16}, K_1^{16}, K_2^{16}, K_3^{16}, K_4^{16}, K_5^{16}, K_6^{16}, K_7^{16}, K_8^{16}, K_9^{16}, K_{10}^{16}, K_{11}^{16}, K_{12}^{16}, K_{13}^{16},$
  $K_{14}^{16}, K_{15}^{16}.$

References

1. Abed, F., List, E., Wenzel, J., Lucks, S.: Differential cryptanalysis of round-reduced simon and speck. In: Cid, C., Rechberger, C. (eds.) FSE 2014. LNCS. Springer (2014, to appear)
2. Alizadeh, J., Alkhzaimi, H.A., Aref, M.R., Bagheri, N., Gauravaram, P., Kumar, A., Lauridsen, M.M., Sanadhya, S.K.: Cryptanalysis of SIMON variants with connections. In: Sadeghi, A.-R., Saxena, N. (eds.) RFIDSec 2014. LNCS, vol. 8651, pp. 90–107. Springer, Heidelberg (2014)
3. Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., Wingers, L.: The SIMON and SPECK Families of Lightweight Block Ciphers. Cryptology ePrint Archive, Report 2013/404 (2013). http://eprint.iacr.org/
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. Biryukov, A., Roy, A., Velichkov, V.: Differential analyis of block cipher SIMON and SPECK. In: Cid, C., Rechberger, C. (eds.) FSE 2014. LNCS. Springer (2014, to appear)
6. 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)
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. Canteaut, A., Naya-Plasencia, M., Vayssière, B.: Sieve-in-the-middle: improved MITM attacks. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013, Part I. LNCS, vol. 8042, pp. 222–240. Springer, Heidelberg (2013)
9. 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)
10. Fuhr, T., Minaud, B.: Match box meet-in-the-middle attack against KATAN. In: FSE 2014. Springer (2014, to appear)
11. 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)
12. Borghoff, J., Canteaut, A., Güneysu, T., Kavun, E.B., Knezevic, M., Knudsen, L.R., Leander, G., Nikov, V., Paar, C., Rechberger, C., Rombouts, P., Thomsen, S.S., Yalçın, T.: PRINCE – a low-latency block cipher for pervasive computing applications. In: Wang, X., Sako, K. (eds.) ASIACRYPT 2012. LNCS, vol. 7658, pp. 208–225. Springer, Heidelberg (2012)
13. 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)
14. 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)
15. Matsui, M.: Linear cryptanalysis method for DES cipher. In: Helleseth, T. (ed.) EUROCRYPT 1993. LNCS, vol. 765, pp. 386–397. Springer, Heidelberg (1994)

16. Shibutani, K., Isobe, T., Hiwatari, H., Mitsuda, A., Akishita, T., Shirai, T.: *Piccolo*: an ultra-lightweight blockcipher. In: Preneel, B., Takagi, T. (eds.) CHES 2011. LNCS, vol. 6917, pp. 342–357. Springer, Heidelberg (2011)
17. Isobe, T., Shibutani, K.: All subkeys recovery attack on block ciphers: extending meet-in-the-middle approach. In: Knudsen, L.R., Wu, H. (eds.) SAC 2012. LNCS, vol. 7707, pp. 202–221. Springer, Heidelberg (2013)
18. Wang, N., Wang, X., Jia, K., Zhao, J.: Improved Differential Attacks on Reduced SIMON Versions. http://eprint.iacr.org/2014/448
19. 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)

Protocols

A Provably Secure Offline RFID Yoking-Proof Protocol with Anonymity

Daisuke Moriyama(✉)

National Institute of Information and Communications Technology, Tokyo, Japan
dmoriyam@nict.go.jp

Abstract. Yoking-proof in a Radio Frequency Identification (RFID) system provides the evidence that two RFID tags are simultaneously scanned by the RFID reader. Though there are numerous yoking-proof protocols, vulnerabilities related to security and privacy are found in many prior works. We introduce a new security definition that covers the man-in-the-middle (MIM) attack, and a privacy definition based on an indistinguishability framework. We also provide a simple construction of a provably secure offline yoking-proof protocol based on the pseudo-random function.

Keywords: RFID · Authentication · Yoking proof · Provable security

1 Introduction

Radio Frequency Identification (RFID) technology enables identification of objects with wireless communication. When a passive RFID tag is attached to the object and electricity is supplied from the RFID reader, a communication channel between the tag and reader is established and they can exchange messages. Since RFID tags are widely used in various commercial activities (e.g., logistics, transportation, product management), tracking of RFID tags by an authorized manager is a fundamental issue of concern. Notably, Juels introduced an RFID yoking-proof protocol in which two RFID tags are simultaneously scanned by the reader and coexistence of the tags is provided by the communication through the reader [13]. When the yoking-proof protocol is generalized so that a group of tags is communicated via the reader and generates a proof by which they all engage in the protocol, it is called a "grouping-proof protocol" [4,22].

Afterwards, various yoking-proof and grouping-proof protocols have been proposed, but almost all of them were broken. Though Juels proposed two yoking-proof protocols, both are vulnerable to replay attacks in which an adversary combines flows from different sessions [4,22]. Piramuthu showed that the grouping proof protocol proposed in [22] is still vulnerable to a replay attack [19]. Subsequently, Peris-Lopez et al. introduced a multi-proof session replay attack in [20] and provided an attack against the Piramuthu's protocol [19]. Burmester et al. proposed a provably secure yoking-proof protocol [3], but Peris et al. pointed out

© Springer International Publishing Switzerland 2015
T. Eisenbarth and E. Öztürk (Eds.): LightSec 2014, LNCS 8898, pp. 155–167, 2015.
DOI: 10.1007/978-3-319-16363-5_10

that their protocol is vulnerable to multiple impersonation attacks [21]. They also mentioned that the yoking-proof protocol proposed by Chien and Liu [6] has a privacy problem and the grouping proof protocol proposed by Huang and Ku [10] is not secure against impersonation attacks. Though Peris-Lopez et al. gave a guideline for constructing secure yoking-proof and grouping-proof protocols and proposed a new yoking-proof protocol, recently Bagheri and Safkhani showed that the tag's secret key can be calculated from the communication message in their protocol [5]. Batina et al. proposed a grouping-proof protocol based on public key cryptography [2], and Hermans and Peeters showed that an impersonation attack and man-in-the-middle (MIM) attack could be launched against it [11].

Our Contributions. In this paper we focus on a provably secure offline RFID yoking-proof protocol. Our contributions are twofold. First, we investigate a rigorous security model for yoking-proof protocols based on the security and privacy definition for canonical RFID authentication protocols [7,15,17]. Though [3] proposed a security model based on the universal composability (UC) framework, it is widely known that it is difficult to provide a security proof in the UC framework. Hermans and Peeters provided another security model in [11] that is based on the existing security model for the canonical RFID authentication protocols [12], but this definition only captures the impersonation attack on security and the adversary cannot learn whether the resulting yoking proof is valid. Different from the above prior security models, our security model covers general MIM attacks in the security definition so that a malicious adversary can interact with all RFID tags at any time and modify the communication. An indistinguishability-based privacy definition is also proposed to provide anonymity for the RFID tags.

Next, we show an example of a yoking-proof protocol that satisfies the proposed security model. Notably, our protocol is robust against an MIM attack that includes all attacks described in previous works [4,11,19,21,22], and satisfies the requirement of anonymity such that no information about the tag is leaked from the communication message. Compared to the existing offline yoking-proof protocol proposed by Hermans and Peeters [11], which has provable security, our protocol does not require public key cryptography and its main building block is a secure pseudorandom function. Thus, our protocol is quite efficient and easier to implement in low-cost RFID tags.

2 Security Model for RFID Yoking-Proof Protocols

Though there are many yoking-proof and grouping-proof protocols, there is no widely known security model and this area is still under development. In this paper, we formalize two basic requirements for yoking-proof protocols, correctness and security. In addition, we also consider the privacy issue as an additional property. Our security model for yoking-proof protocols is motivated by the security model for basic RFID authentication protocols [15]. However, we do not intentionally cover the tag corruption in the following definition because almost all previous works does not satisfy a classical man-in-the-middle attack nor privacy.

2.1 Execution Model

A yoking-proof protocol in an RFID system is executed among the verifier $\mathcal{V}$, RFID reader $\mathcal{R}$ and multiple RFID tags in $\mathcal{T} := \{t_0, t_1, \ldots, t_n\}$. The yoking-proof protocol consists of three phases: setup, generation and verification. In the setup phase, the verifier runs a setup algorithm with security parameter k and obtains public parameter and secret keys. If the protocol is based on symmetric-key cryptography, the verifier shares the secret key with the RFID tags. In the generation phase, two RFID tags generate a yoking-proof via the reader. Then, two RFID tags in the same group execute an interactive protocol and finally the reader outputs a proof. The verifier checks the validity of the proof in the verification phase. If the verification is accepted, the verifier outputs 1 (acceptance); otherwise, it outputs 0 (rejection) as the verification result. In the following, we concentrate on the offline yoking-proof protocol wherein the verifier does not participate in the generation phase.

We consider that each session of the party (RFID reader and individual tags) is identified by a session identifier sid, that contains the collected of the input/output messages of the party. A session is called finished when the party outputs the final message in the session. We say that a party A has a matching session with the other party B if all communication messages between the two are honestly transferred. The correctness of the yoking-proof protocol is that the verifier always accepts the yoking proof if it is generated by the session wherein two tags in the same group have a matching session with the peer.

We note that the role of the reader is different from the canonical RFID authentication protocol in which the RFID reader authenticates the RFID tag and outputs the authentication result. Therefore, the RFID reader actively participates in the protocol. On the other hand, the reader in the yoking-proof and grouping-proof protocols does not authenticate tags[1] and its task is to transfer the message among tags and obtain the yoking proof from the communication message. Instead, a verifier checks the validity of the yoking proof after a session is finished and the reader submits the yoking proof. Thus it is trivial that an adversary impersonates a reader and this is not a security issue in yoking-proof and grouping-proof protocols. Therefore, the protocol procedure is quite different and the existing security model for typical RFID authentication protocols is not directly applicable to the yoking-proof protocols.

2.2 Security

Intuitively, the security of the yoking-proof protocol requires that the verifier always rejects the yoking proof if it is not available in one honest protocol execution between the RFID tags. When there is an active adversary that can interfere, delay, interleave and modify the communication message, we can consider there are two security levels: resistance against the impersonation attack and

[1] Tag authentication by the reader can be one application, but it is not a necessary issue in yoking-proof protocols.

MIM attack. Hermans and Peeters [11] introduced a formal security model for the RFID yoking-proof protocol that captures the impersonation attack. In their security definition, the adversary can communicate with all tags in the learning phase. After that, the adversary cannot interact with one of the uncorrupted tags and the goal of the adversary is to impersonate the tag and output a valid yoking proof. Though the above impersonation resistance is also widely known as an active attack, it does not imply security against an MIM attack. Notably, several lightweight authentication protocols [1,8,14] are provably secure against an active adversary but vulnerable to a (general) MIM attack [8,9,18]. In this paper, we formalize security against a general MIM attack in RFID yoking-proof protocols.

When a general MIM attack is launched, the adversary can interact with all tags at any time. The adversary's goal is to output a valid yoking proof that is not generated in the sessions wherein the tag has a matching session to the other tag. We note that the above condition does not mean that the reader has no matching session to one of the RFID tags. Whereas the reader always transmits the communication messages between the RFID tags, the adversary can directly deliver the tag's output to the other tag. Even when the adversary sends an arbitrary random message to the reader in the session and the reader does not have a matching session, the adversary can obtain a valid yoking proof derived from the tags.

More formally, we provide the following general security experiment $\mathsf{Exp}^{\mathsf{Sec}}_{\Pi,\mathcal{A}}(k)$ between a challenger and adversary $\mathcal{A}$ against an RFID yoking-proof protocol Π.

Setup. The challenger runs the setup algorithm and provides a public parameter to the adversary.

Learning. Then $\mathcal{A}$ can then adaptively issue the following queries to interact with the reader, tags and verifier:
- $\mathsf{Launch}(1^k)$: Launch the reader to start a new session.
- $\mathsf{SendReader}(m)$: Send an arbitrary message m to the reader.
- $\mathsf{SendTag}(t, m)$: Send an arbitrary message m to the tag $t \in \mathcal{T}$.
- $\mathsf{Result}(\sigma)$: Output whether the verifier accepts the yoking-proof σ.

Guess. When the adversary finishes the interaction $\mathcal{A}$ outputs (t_0^*, t_1^*, σ^*) where $(t_0^*, t_1^*) \subseteq \mathcal{T}$. The challenger outputs 1 if $\mathsf{Result}(\sigma^*) = 1$ and σ^* is not derived from the session in which t_0^* has the matching session to t_1^*. Otherwise, the challenger outputs 0.

The probability that the adversary wins the above security game is denoted by $\mathsf{Adv}^{\mathsf{Sec}}_{\Pi,\mathcal{A}}(k) := \Pr[\mathsf{Exp}^{\mathsf{Sec}}_{\Pi,\mathcal{A}}(k) \to 1]$.

Definition 1. *An RFID yoking-proof protocol Π is secure against general MIM attacks if for any probabilistic polynomial time adversary $\mathcal{A}$, $\mathsf{Adv}^{\mathsf{Sec}}_{\Pi,\mathcal{A}}(k)$ is negligible in k.*

We note that a general MIM attack covers all previous attacks against previous works [4,11,19,21,22]. One typical approach of replay attacks is that a malicious adversary captures the past communication messages and reuses them in

the target session. In the impersonation attack the adversary sends adversarial messages to the tag and obtains a meaningful information. Our security definition does not restrict any strategy from the view point of the adversary and only gives one exception wherein σ^* honestly generated by two tags in one session cannot be submitted since it is trivial in terms of the correctness property.

Different from the existing security definition for RFID authentication protocols, the final goal of the adversary is not to submit an acceptable message to the reader but to output a forged yoking proof to the offline verifier. We therefore define the above experiment as the security definition against digital signature or message authentication code schemes.

2.3 Privacy

One application of the RFID yoking-proof protocol is to cover anonymity. Though the yoking-proof system is useful to prove when the tag interacts with the reader from the view point of the honest verifier, it is not desirable that a malicious adversary can trace the tag. Even if the adversary cannot verify the yoking proof two tags generate, several previous works specify that the tag explicitly outputs its identity to the reader, so the adversary can learn which tag tries to generate a yoking proof. Since the tag's anonymity is a critical issue in RFID authentication protocols, it is useful to consider privacy in the related topics.

In this paper we formalize the privacy definition for RFID yoking-proof protocols based on indistinguishability-based privacy for canonical RFID authentication protocols [7, 15, 17]. Consider the following privacy experiment against an RFID yoking-proof protocol Π between a challenger and adversary $\mathcal{A} := (\mathcal{A}_1, \mathcal{A}_2)$:

Setup. The challenger runs the setup algorithm to initialize the verifier, reader and tag. The adversary obtains public parameter pp and the identities of the reader and tag $(\mathcal{R}, \mathcal{T})$.

Phase 1. The adversary $\mathcal{A}_1$ can issue $\mathcal{O}:=\{\mathsf{Launch}, \mathsf{SendReader}, \mathsf{SendTag}, \mathsf{Result}\}$ to communicate with the reader and tag and check the validity of the yoking proof as a security game.

Challenge. $\mathcal{A}$ sends two sets of tags $\mathcal{T}_0^*$ and $\mathcal{T}_1^*$ $(\mathcal{T}_0^* \neq \mathcal{T}_1^* \ \wedge \ |\mathcal{T}_0^*| = |\mathcal{T}_1^*| = 2)$ to the challenger and outputs state information st. The challenger flips a coin $b \overset{\mathsf{U}}{\leftarrow} \{0, 1\}$ and sets $\mathcal{T}' := \mathcal{T} \setminus \{\mathcal{T}_0^*, \mathcal{T}_1^*\}$.

Phase 2. The adversary $\mathcal{A}_2$ receives st and continues to interact with $\mathcal{R}$ and $\mathcal{T}'$ as Phase 1. If the adversary wants to send message m to a tag in the group $\mathcal{T}_b^*$, he issues $\mathsf{SendTag}((\mathcal{I}, i), m)$ with an intermediate algorithm $\mathcal{I}$. $\mathcal{I}$ relays the communication between $\mathcal{A}_2$ and the i-th member of $\mathcal{T}_b^*$ to prevent the adversary from directly interacting with the target tag.

Guess. Finally, the adversary $\mathcal{A}_2$ outputs b'.

The adversary wins the above game if $b' = b$ holds and two tags in $\mathcal{T}_0^*/\mathcal{T}_1^*$ belong to the same group. Depending on the flipped coin b, we define the above experiment as $\mathsf{Exp}^{\mathsf{IND}}_{\Pi,\mathcal{A}}b(k)$. Then the advantage of the adversary is defined by

$$\mathsf{Adv}^{\mathsf{IND}}_{\Pi,\mathcal{A}}(k) = \left| \Pr[\mathsf{Exp}^{\mathsf{IND\text{-}0}}_{\Pi,\mathcal{A}}(k) \to 1] - \Pr[\mathsf{Exp}^{\mathsf{IND\text{-}1}}_{\Pi,\mathcal{A}}(k) \to 1] \right|.$$

Definition 2. *An RFID yoking-proof protocol Π satisfies IND-privacy if for any PPT adversary $\mathcal{A}$, $\mathsf{Adv}^{\mathsf{IND}}_{\Pi,\mathcal{A}}(k)$ is negligible in k.*

Recall that $\mathcal{T}$ contains identities of all RFID tags. Since there are many groups in $\mathcal{T}$, the adversary may want to distinguish between the group of RFID tags. Consider that there are two groups $A \subset \mathcal{T}$ and $B \subset \mathcal{T}$. If $\mathcal{T}_0^* \subseteq A$ and $\mathcal{T}_1^* \subseteq B$, the above definition claims that the adversary cannot distinguish between two groups even when it obtains the communication messages anonymously. On the other hand, if $\mathcal{T}_0^* \subset A$ and $\mathcal{T}_1^* \subset A$ for a group A, this means that the adversary tries to distinguish the tag frin among the members in A; thus, no information related to the group's or tag's identity should be leaked from the communication messages to satisfy the requirement of IND-privacy.

In this paper we do not formalize the forward secrecy so that a malicious adversary corrupts the RFID tag and obtains the internal secret. Since we focus on the offline yoking-proof protocol, it is not trivial for RFID tags to securely update the secret key with the offline verifier. The key updating mechanism is certainly an additional issue, and we leave this issue as an open problem. Since almost all the previous symmetric-key based protocols are broken and the security level of the existing provably secure yoking-proof protocol [11] is only an impersonation attack, the first task is to show a yoking-proof protocol provably secure against a general MIM attack.

3 Proposed Yoking-Proof Protocol

We present a new RFID yoking-proof protocol, provably secure against a general MIM attack and satisfies IND-privacy. It is based on the previous yoking-proof protocol proposed by Bermester et al. [3] and supports group authentication during yoking-proof generation. Assume that a secure pseudorandom function $\mathsf{PRF} : \{0,1\}^k \times \{0,1\}^* \to \{0,1\}^k$ is implemented in all RFID tags. The proposed protocol proceeds as follows:

Setup Phase. The verifier $\mathcal{V}$ randomly selects group secret key $k_X \stackrel{\mathsf{U}}{\leftarrow} \{0,1\}^k$ for each group and individual secret key $k_i \stackrel{\mathsf{U}}{\leftarrow} \{0,1\}^k$ for each tag. Tag t_i receives (k_X, k_i) from the verifier. Reader $\mathcal{R}$ selects a maximum delay time δ that denotes the upper bound of the execution for each session.

Generation Phase. For simplicity, we consider that tag t_1 and t_2 communicate with each other to generate the yoking proof.

1. Reader $\mathcal{R}$ sends time stamp ts and random nonce $r \stackrel{\mathsf{U}}{\leftarrow} \{0,1\}^k$ to tag t_1. $\mathcal{R}$ also starts an internal time clock.
2. Upon receiving (ts, r) from $\mathcal{R}$, t_1 randomly chooses $r_1 \stackrel{\mathsf{U}}{\leftarrow} \{0,1\}^k$ and computes $u_1 := \mathsf{PRF}(k_X, (\mathsf{ts}, r, r_1))$. t_1 sends (r, r_1, u_1) to $\mathcal{R}$.
3. $\mathcal{R}$ sends $(\mathsf{ts}, r, r_1, u_1)$ to t_2.
4. Upon receiving $(\mathsf{ts}, r, r_1, u_1)$ from $\mathcal{R}$, t_2 verifies $u_1 = \mathsf{PRF}(k_X, (\mathsf{ts}, r, r_1))$. If this verification holds, t_2 randomly chooses $r_2 \stackrel{\mathsf{U}}{\leftarrow} \{0,1\}^k$ and computes

$u_2 := \mathsf{PRF}(k_X, (\mathsf{ts}, r, r_1, r_2))$ and $v_2 := \mathsf{PRF}(k_2, (\mathsf{ts}, r, r_1, r_2))$. Otherwise, t_2 randomly selects $(r_2, u_2, v_2) \stackrel{\mathsf{U}}{\leftarrow} \{0,1\}^{3k}$. t_2 sends (r, r_2, u_2, v_2) to $\mathcal{R}$.

5. $\mathcal{R}$ sends (r_1, r_2, u_2) to t_1.
6. Upon receiving (r_1, r_2, u_2) from $\mathcal{R}$, t_1 checks that there is an unfinished session where t_1 outputs r_1 in the first round and verifies $u_2 := \mathsf{PRF}(k_X, (\mathsf{ts}, r, r_1, r_2))$. If this verification holds, t_1 computes $v_1 := \mathsf{PRF}(k_1, (\mathsf{ts}, r, r_1, r_2))$. Otherwise, t_1 selects $v_1 \stackrel{\mathsf{U}}{\leftarrow} \{0,1\}^k$. t_1 sends (r, v_1) to $\mathcal{R}$.
7. $\mathcal{R}$ outputs $\sigma := (\mathsf{ts}, r, r_1, r_2, u_2, v_1, v_2)$ as a yoking proof if the above execution is finished within δ. Otherwise, $\mathcal{R}$ aborts the session.

Verification Phase. The verifier checks $u_2 = \mathsf{PRF}(k_X, (\mathsf{ts}, r, r_1, r_2))$ for a group secret key k_X. If this verification holds, $\mathcal{V}$ checks $v_1 = \mathsf{PRF}(k_1, (\mathsf{ts}, r, r_1, r_2))$ and $v_2 = \mathsf{PRF}(k_2, (\mathsf{ts}, r, r_1, r_2))$ for some individual secret keys k_1 and k_2 sent to the group member. If these verifications hold, $\mathcal{V}$ accepts the yoking proof and outputs 1. If one of the three verifications fails, $\mathcal{V}$ outputs 0.

The main building block of our protocol is the pseudorandom function. Because it is natural for the existing RFID authentication protocols that the RFID tag can compute symmetric key cryptography, we do not restrict here that the computational resource of RFID tags must be EPC-compliant. In particular, it is quite hard to provide a security proof for EPC-compliant protocols because such tags can only execute basic operations such as AND, OR, XOR, or cyclic-shift. Though we extend a protocol in [3] that is vulnerable to the multiple impersonation attacks [21], our protocol specifies that (u_2, v_1, v_2) is computed with fresh nonces selected by each tag and an impersonation attack always fails. One of the two nonces is always selected by itself and the other nonce restricts the peer of the session; so our protocol is secure against a general MIM attack (see more detailed discussion in the next section).

If the tag's identity is also input to PRF, each tag can verify which member of the group executes the yoking-proof protocol. However, it is inefficient for the tag to run an exhaustive search to check the actual peer in the group when the group member is quite large (such as for the role of the reader in the canonical RFID authentication protocol). Even when the tag's identity is not included, each tag can check whether a valid tag in the same group tries to execute the yoking proof in our protocol.

In the above protocol, time stamp ts and nonce r are always input to the pseudorandom function. The role of the time stamp is to specify when the reader invokes the session and the verifier learns it, while the time stamp is useless for the communicating RFID tags. On the other hand, r is used to ensure the concurrent execution of the session. Especially, Lin et al. [16] considered the situation in which the tag communicates with multiple readers and executes many sessions concurrently. When some readers start different sessions at the same time, the time stamp may be recorded. However, the random nonce r is also chosen at random and is unique for each reader (the collision probability is at most 2^{-k}). Therefore, the communication message output from each tag contains r to distinguish between multiple readers. Similarly, t_1 receives r_1 from $\mathcal{R}$ even after r_1 is sent to $\mathcal{R}$. Different from tag t_2, t_1 starts a new session

whenever it receives (ts, r) and must keep sessions until the reader gives the response from t_2 to t_1. If we omit r_1 from (r_1, r_2, u_2) transmitted from $\mathcal{R}$, t_1 cannot learn which session should be verified in the concurrent execution, so some extra information to distinguish multiple sessions is needed to transfer messages. Since (r, r_1) is chosen randomly for each session and can be used as unique identifier, these nonces are contained in the communication message to support concurrent execution.

We note that the validity of v_1 and v_2 cannot be checked by the tags in our protocol and one may think that there is a chance for a malicious adversary to change these variables. Actually, the tags cannot detect the man-in-the-middle attack. But v_1 and v_2 are used to check by the (legitimate) verifier and these values are strictly determined by the time stamp and nonces related to the session. Since the goal of the adversary is to violate the security or privacy as explained in the previous section, no critical problem occurs in our protocol.

4 Security Proof

We prove that our protocol satisfies the security model described in Sect. 2. It is clear that the proposed protocol described in the previous section satisfies correctness.

Theorem 1. *Our yoking-proof protocol is secure against a general MIM attack if* PRF *is a secure pseudorandom function.*

Proof. We show that if an adversary $\mathcal{A}$ wins the security game described in Sect. 2, there is an algorithm $\mathcal{B}$ that breaks the security of pseudorandom function PRF. $\mathcal{B}$ can issue oracle queries to the function that is actual pseudorandom function $\mathsf{PRF}(k_1, \cdot)$ or a truly random function, and the goal of $\mathcal{B}$ is to distinguish between them. Consider that the adversary outputs a set of the tag's identities (t_1^*, t_2^*) and yoking proof $\sigma^* := (r^*, r_1^*, r_2^*, u_2^*, v_1^*, v_2^*)$. To satisfy $\mathsf{Result}(\sigma^*) = 1$, all verifications must be passed so that $u_2^* = \mathsf{PRF}(k_X, (r^*, r_1^*, r_2^*))$, $v_1^* = \mathsf{PRF}(k_1^*, (r^*, r_1^*, r_2^*))$ and $v_2^* = \mathsf{PRF}(k_2^*, (r^*, r_1^*, r_2^*))$ where (k_1^*, k_2^*) is a secret key of (t_1^*, t_2^*), respectively. On the other hand, t_1^* does not have matching session to t_2^* in the related session and the adversary cannot simply forward the communication message between these tags. The strategy of the adversary is divided into two cases:

Case 1: $\mathcal{A}$ outputs a valid v_1^* while v_1^* does not appear in the t_1^*'s output.
Case 2: $\mathcal{A}$ outputs v_1^* derived from t_1^*.

When Case 1 occurs, we can construct an algorithm $\mathcal{B}$ that breaks the security of pseudorandom function $\mathsf{PRF}(k_1, \cdot)$. $\mathcal{B}$ internally runs $\mathcal{A}$ and simulates the above protocol. $\mathcal{B}$ selects all group secret keys and individual secret keys except for t_1^*'s secret key. $\mathcal{B}$ honestly simulates all communication messages except the case that t_1^* is activated and it outputs a final message in the session. When $\mathcal{A}$ issues $\mathsf{SendTag}(t_1^*, (r_1, r_2, u_2))$, $\mathcal{B}$ checks $u_2 = \mathsf{PRF}(k_X, (r, r_1, r_2))$ and sends (r, r_1, r_2)

to the challenger of the pseudorandom function. When $\mathcal{B}$ receives the response v_1 from the challenger, $\mathcal{B}$ transfers (r, v_1) to $\mathcal{A}$. When $\mathcal{A}$ outputs a forged yoking-proof σ^*, $\mathcal{B}$ issues (r^*, r_1^*, r_2^*) to the challenger and obtains v^*. If $v^* = v_1^*$, $\mathcal{B}$ outputs 1 and halts the simulation. Otherwise, $\mathcal{B}$ outputs 0.

If $\mathcal{B}$ interacts with the actual pseudorandom function $\mathsf{PRF}(k_1^*, \cdot)$, the probability $v^* = v_1^*$ holds is non-negligible since we assume that $\mathcal{A}$ outputs a valid yoking proof. Thus $\mathcal{B}$ outputs 1 with non-negligible probability. On the other hand, if the challenger selects a truly random function, it is impossible to guess the valid output and $\mathcal{B}$ outputs 1 with negligible probability $1/2^{-k}$. Therefore $\mathcal{B}$ can break the security of the pseudorandom function.

We note that the same argument can be applied to v_2^*. If v_2^* is not honestly generated by t_2^*, the adversary cannot output a valid yoking-proof based on the security of the pseudorandom function. On the other hand, t_2^* outputs $v_2^* = \mathsf{PRF}(k_X, (\mathsf{ts}^*, r^*, r_1^*, r_2^*))$ if and only if t_2^* receives $(\mathsf{ts}^*, r^*, r_1^*, u_1^*)$ and selects r_2^*. Since (u_2^*, v_1^*, v_2^*) is checked with the same input $(\mathsf{ts}^*, r^*, r_1^*, r_2^*)$ and deterministically defined, the adversary cannot output forged yoking-proof that is not output by (t_1^*, t_2^*). □

One may think that the verification check by t_2 in Fig. 1 can be passed even when the adversary issues another tag t_3 in the same group and transfers the message (r, r_1, u_1). However, (u_2, v_2) is computed with random nonces r_1, r_2 that specify that only the party that generates r_1 accepts the session with verification check of u_2. t_1 clearly rejects the session when the adversary sends (r_1, r_2, u_2) to t_1, so u_2 binds the participants of the session. Similarly, while t_1 also accepts any messages generated by the same group, the output v_1 for each session rigorously

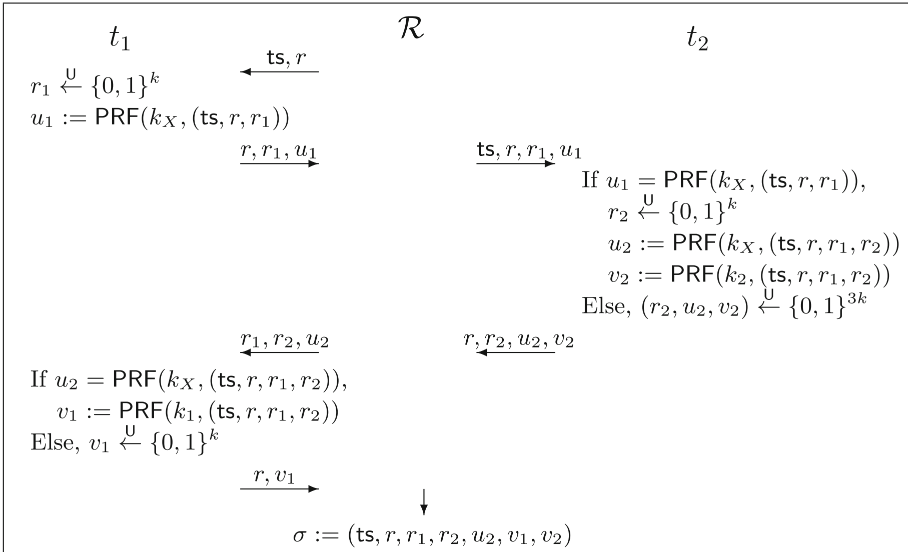

Fig. 1. The proposed yoking-proof protocol

restricts that the peer honestly receives r_1 (generated by t_1 itself) and validity check by the verifier is accepted only if (u_2, v_2) and v_1 are computed by the same input. Therefore, even if the adversary transfers the communication message used in a different session, the verifier always rejects the yoking proof.

Theorem 2. *Our protocol satisfies IND-privacy if* PRF *is a secure pseudorandom function.*

Proof. Intuitively, communication messages derived from the tag in our protocol consist of only random nonces and outputs from the pseudorandom function. In addition, fresh nonces are selected for each individual session and the adversary cannot find any correlation between the sessions (e.g., whether the same tag executes the two sessions). We show the formal security proof of our protocol with the game transformation technique. Recall that n denotes the number of RFID tags in the system. We consider that the number of group in the yoking-proof protocol is denoted as n'.

Game 0. This is the original privacy experiment between a challenger and adversary $\mathcal{A}$.

Game 1-j. For each $1 \leq i \leq j$, the challenger assigns random variables $(u_1, u_2) \xleftarrow{U} \{0,1\}^k$ instead of $\mathsf{PRF}(k_X, \cdot)$ that i-th group member computes for any session.

Game 2-j. For each $1 \leq i \leq j$, the challenger assigns random variable $v_i \xleftarrow{U} \{0,1\}^k$ instead of the t_i's computation $\mathsf{PRF}(k_i, \cdot)$.

Let S_i be a probability that the adversary wins the privacy experiment in Game i.

Lemma 1. *We have* $|S_{1\text{-}(j-1)} - S_{1\text{-}j}| \leq \mathsf{Adv}_{\mathcal{B}}^{\mathsf{PRF}}(k)$.

If the final output of the adversary $\mathcal{A}$ is different between Game 1-$(j-1)$ and Game 1-j, there is an algorithm $\mathcal{B}$ that breaks the security of the pseudorandom function. $\mathcal{B}$ generates all secret keys except k_X and internally runs $\mathcal{A}$. If an activated tag belongs to the i-th group where $i < j$, $\mathcal{B}$ always selects random variables $(u_1, u_2) \xleftarrow{U} \{0,1\}^k$ instead of computing pseudorandom function and proceeds with the simulation. If an activated tag belongs to the i-th group where $i > j$, $\mathcal{B}$ honestly computes (u_1, u_2) with an actual pseudorandom function $\mathsf{PRF}(k_{X'}, \cdot)$ where $K_{X'}$ is a group secret key of the i-th group. When one of the members of j-th group is activated with input r, $\mathcal{B}$ proceeds as follows. $\mathcal{B}$ selects $r_1 \xleftarrow{U} \{0,1\}^k$ and sends (ts, r, r_1) to the challenger. Upon receiving u_1 from the challenger, $\mathcal{B}$ sets (r, r_1, u_1) as the tag's output. When $\mathcal{A}$ sends (ts, r, r_1, u) to a member of the i-th group, $\mathcal{B}$ issues (ts, r, r_1) to the challenger and compares its response with u. If it holds, $\mathcal{B}$ selects $r_2 \xleftarrow{U} \{0,1\}^k$, issues $(\mathsf{ts}, r, r_1, r_2)$ to the challenger and obtains u_2. v_2 can be computed since $\mathcal{B}$ chooses an individual secret key by itself and (r, r_2, u_2, v_2) is assigned as the output from the tag. When $\mathcal{A}$ sends (r_1', r_2', u_2') to a member of the i-th group, $\mathcal{B}$ checks whether the tag previously outputs r_1'. If so, $\mathcal{B}$ finds the corresponding nonce r and issues (r, r_1', r_2') to the oracle to compare the response with u_2'. $\mathcal{B}$ proceeds with the

above simulation regardless of the choice of the two sets of tags $\mathcal{A}$ sends in the challenge phase. When $\mathcal{A}$ outputs a bit b, $\mathcal{B}$ stops the simulation and outputs the same bit.

If the challenger gives actual pseudorandom function $\mathsf{PRF}(k_X, \cdot)$ to $\mathcal{B}$, the above simulation is equivalent to Game 1-$(j-1)$. Otherwise, if $\mathcal{B}$ interacts with truly random function, the outputs of the j-th member are coming from the random function and it is equivalent to Game 1-j. Therefore, if $\mathcal{A}$ distinguishes the difference between Game 1-$(j-1)$ and Game 1-j, $\mathcal{B}$ can break the security of the pseudorandom function PRF.

Lemma 2. *We have* $|S_{2\text{-}(j-1)} - S_{2\text{-}j}| \leq \mathsf{Adv}_{\mathcal{B}}^{\mathsf{PRF}}(k)$.

The proof strategy of Lemma 2 is analogous to the proof of Lemma 1 so we omit the details. When tag t_j is activated we replace the communication message derived from the pseudorandom function to random strings. If an adversary finds a gap between these games, the security of the pseudorandom function can be broken.

Since Game 0 and Game 1-n' can be considered as Game 1-0 and Game 2-0 respectively, we can transform Game 0 to Game 2-n based on the security of the pseudorandom function. When we proceed with Game 2-n, there is no chance for the adversary to distinguish between the RFID tags. In this game, all communication messages generated by the tags in $\mathcal{T}$ consist of nonces freshly chosen per session and random strings derived from a truly random function, so no information about the tag's identity is observed in the communication message (including anonymous interaction). We can therefore say that $S_{2\text{-}n} = 0$.

Eventually, we have $\mathsf{Adv}_{\Pi,\mathcal{A}}^{\mathsf{IND}}(k) \leq (n' + n) \cdot \mathsf{Adv}_{\mathcal{B}}^{\mathsf{PRF}}(k)$. □

5 Conclusions and Future Work

In this paper, we introduced a new security model for an RFID yoking-proof protocol. Our model formalizes the security against a general MIM attack that covers all previous attacks for yoking-proof protocols. The indisinguishability-based privacy is also defined, which captures the RFID tags's anonymity. The example protocol given in this paper does not require public key cryptography and is simply described with a secure pseudorandom function.

Since we focus on the basic provably secure offline yoking-proof protocol based on the symmetric key primitive, the possibility of the key exposure problem of the RFID tag is ignored in this paper. Since the verifier is offline and cannot participate in the yoking-proof generation, updating the shared secret with the RFID tag to satisfy the requirement of forward secrecy is an open problem. Another problem tag corruption causes will be an impersonation attack by the corrupted tag in the same group.

References

1. Bringer, J., Chabanne, H., Dottax, E.: HB++: a lightweight authentication protocol secure against some attacks. In: SecPerU 2006, pp. 28–33 (2006)
2. Batina, L., Lee, Y.K., Seys, S., Singelée, D., Verbauwhede, I.: Extending ECC-based RFID authentication protocols to privacy-preserving multi-party grouping proofs. J. Pers. Ubiquit. Comput. **16**(3), 323–335 (2012). Springer, Heidelberg
3. Burmester, M., de Medeiros, B., Motta, R.: Provably secure grouping-proofs for RFID tags. In: Grimaud, G., Standaert, F.-X. (eds.) CARDIS 2008. LNCS, vol. 5189, pp. 176–190. Springer, Heidelberg (2008)
4. Bolotnyy, L., Robins, G.: Generalized "yoking-proofs" for a groupe of RFID tags. In: Mobiquitous 2006, pp. 1–4. IEEE (2006)
5. Bagheri, N., Safkhani, M.: Securet disclosure attack on Kazahaya, a yoking-proof for low-cost RFID tags. Cryptology ePrint Archive, Report 2013/453
6. Chien, H.-Y., Liu, S.-B.: Tree-based RFID yoking proof. In: NSWCTC 2009, pp. 550–553. IEEE (2009)
7. Deng, R.H., Li, Y., Yung, M., Zhao, Y.: A new framework for RFID privacy. In: Gritzalis, D., Preneel, B., Theoharidou, M. (eds.) ESORICS 2010. LNCS, vol. 6345, pp. 1–18. Springer, Heidelberg (2010)
8. Gilbert, H., Robshaw, M., Sibert, H.: An active attack against HB+ - a provably secure lightweight authentication protocol. IEEE Electron. Lett. **41**(21), 1169–1170 (2005). IEEE
9. Gilbert, H., Robshaw, M., Seurin, Y.: Good variants of HB^+ are hard to find. In: Tsudik, G. (ed.) FC 2008. LNCS, vol. 5143, pp. 156–170. Springer, Heidelberg (2008)
10. Huang, H.-H., Ku, C.-Y.: A RFID grouping proof protocol for medication sefety of inpatient. J. Med. Syst. **33**(6), 467–474 (2009). Springer, Heidelberg
11. Hermans, J., Peeters, R.: Private yoking proofs: attacks, models and new provable constructions. In: Hoepman, J.-H., Verbauwhede, I. (eds.) RFIDSec 2012. LNCS, vol. 7739, pp. 96–108. Springer, Heidelberg (2013)
12. Hermans, J., Pashalidis, A., Vercauteren, F., Preneel, B.: A new RFID privacy model. In: Atluri, V., Diaz, C. (eds.) ESORICS 2011. LNCS, vol. 6879, pp. 568–587. Springer, Heidelberg (2011)
13. Juels, A.: "Yoking-proofs" for RFID tags. In: PerSec 2004, pp. 138–143. IEEE (2004)
14. Juels, A., Weis, S.A.: Authenticating pervasive devices with human protocols. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 293–308. Springer, Heidelberg (2005)
15. Juels, A., Weis, S.A.: Defining strong privacy for RFID. ACM TISSEC **13**(1), 7 (2009). ACM
16. Lin, C.-C., Lai, Y.-C., Tygar, J.D., Yang, C.-K., Chiang, C.-L.: Coexistence proof using chain of timestamps for multiple RFID tags. In: Chang, K.C.-C., Wang, W., Chen, L., Ellis, C.A., Hsu, C.-H., Tsoi, A.C., Wang, H. (eds.) APWeb/WAIM 2007 Ws. LNCS, vol. 4537, pp. 634–643. Springer, Heidelberg (2007)
17. Moriyama, D., Matsuo, S., Ohkubo, M.: Relations among notions of privacy for RFID authentication protocols. In: Foresti, S., Yung, M., Martinelli, F. (eds.) ESORICS 2012. LNCS, vol. 7459, pp. 661–678. Springer, Heidelberg (2012)
18. Ouafi, K., Overbeck, R., Vaudenay, S.: On the security of HB# against a man-in-the-middle attack. In: Pieprzyk, J. (ed.) ASIACRYPT 2008. LNCS, vol. 5350, pp. 108–124. Springer, Heidelberg (2008)

19. Piramuthu, S.: On existence proofs for multiple RFID tags. In: PerSecU 2006, pp. 317–328. IEEE (2006)
20. Peris-Lopez, P., Hernandez-Castro, J.C., Estevez-Tapiador, J.M., Ribagorda, A.: Solving the simulutaneous scanning proglem anonymously: clumping proofs for RFID tags. In: SecPerU 2007, pp. 55–60. IEEE (2007)
21. Peris-Lopez, P., Orfila, A., Hernandez-Castro, J.C., Lubbe, J.C.A.: Flaws on RFID grouping-proofs. guidelines for future sound protocols. J. Netw. Comput. Appl. **34**(3), 833–845 (2011). Academic Press
22. Saito, J., Sakurai, K.: Grouping proof for RIFD tags. In: AINA 2005, vol. 2. pp. 621–624. IEEE (2005)

Author Index

AlTawy, Riham 126
Aysu, Aydin 34

Beaulieu, Ray 3

De Santis, Fabrizio 85

Guillen, Oscar M. 85
Gulcan, Ege 34

Hu, Lei 140

Ma, Bingke 140
Makarim, Rusydi H. 109
Matsui, Mitsuru 51
Moriyama, Daisuke 155
Murakami, Yumiko 51

Özbudak, Ferruh 69

Peralta, René 21

Sakic, Ermin 85
Schaumont, Patrick 34
Shi, Danping 140
Shors, Douglas 3
Sigl, Georg 85
Smith, Jason 3
Song, Ling 140

Tezcan, Cihangir 69, 109
Treatman-Clark, Stefan 3
Turan Sönmez, Meltem 21

Weeks, Bryan 3
Wingers, Louis 3

Youssef, Amr M. 126