Analysis of (Lightweight) Symmetric-Key Algorithms and Their Software Implementations

by

Mabin Joseph Puthiakulangara

ENGG02201404006

Indira Gandhi Centre for Atomic Research, Kalpakkam, India

A thesis submitted to the
Board of Studies in Engineering Sciences

In partial fulfillment of requirements
for the Degree of

DOCTOR OF PHILOSOPHY

of

HOMI BHABHA NATIONAL INSTITUTE

July, 2022

Homi Bhabha National Institute

Recommendations of the Viva Voce Committee

As members of the Viva Voce Committee, we certify that we have read the dissertation prepared by Mr. Mabin Joseph Puthiakulangara titled "**Analysis of (Lightweight) Symmetric-Key Algorithms and Their Software Implementations**" and recommend that it may be accepted as fulfilling the thesis requirement for the award of Degree of Doctor of Philosophy.

Chairman	Dr. B.P.C. Rao	
Guide / Convener	Prof. Dr. R. Balasubramanian	
Examiner	Prof. Dr. Subhamoy Maitra *ISI, Kolkata*	
Member 1	Dr. Sharat Chandra	
Member 2	Dr. V.S. Srinivasan	
Technology Adviser	Dr. Gautham Sekar *Madras School of Economics &* *Madras Fintech Services Pvt. Ltd*	

Final approval and acceptance of this thesis is contingent upon the candidate's submission of the final copies of the thesis to HBNI.

I hereby certify that I have read this thesis prepared under my direction and recommend that it may be accepted as fulfilling the thesis requirement.

Date: 23rd May 2023

Place: IGCAR, Kalpakkam

Prof. Dr. R. Balasubramanian

Statement by Author

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at Homi Bhabha National Institute (HBNI) and is deposited in the Library to be made available to borrowers under rules of the HBNI. Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgement of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the Competent Authority of HBNI when in his or her judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

Mabin Joseph Puthiakulangara

Declaration

I, hereby declare that the investigation presented in the thesis has been carried out by me. The work is original and has not been submitted earlier as a whole or in part for a degree / diploma at this or any other Institution / University.

Mabin Joseph Puthiakulangara

List of Publications

Journals

(a) "Side channel analysis of SPECK", Joseph, M., Sekar, G., Balasubramanian, R., *Journal of Computer Security*, **2020**, *28(6)*, 655–676.

(b) "On the Security of the Stream Ciphers RCR-64 and RCR-32", Joseph, M., Sekar, G., Balasubramanian, R., Venkiteswaran, G., *The Computer Journal*, **2021**, *65(12)*, 3091–3099.

(c) "Revisiting the Software-Efficient Stream Ciphers RCR-64 and RCR-32", Joseph, M., Sekar, G., Balasubramanian, R., *The Computer Journal*, to appear.

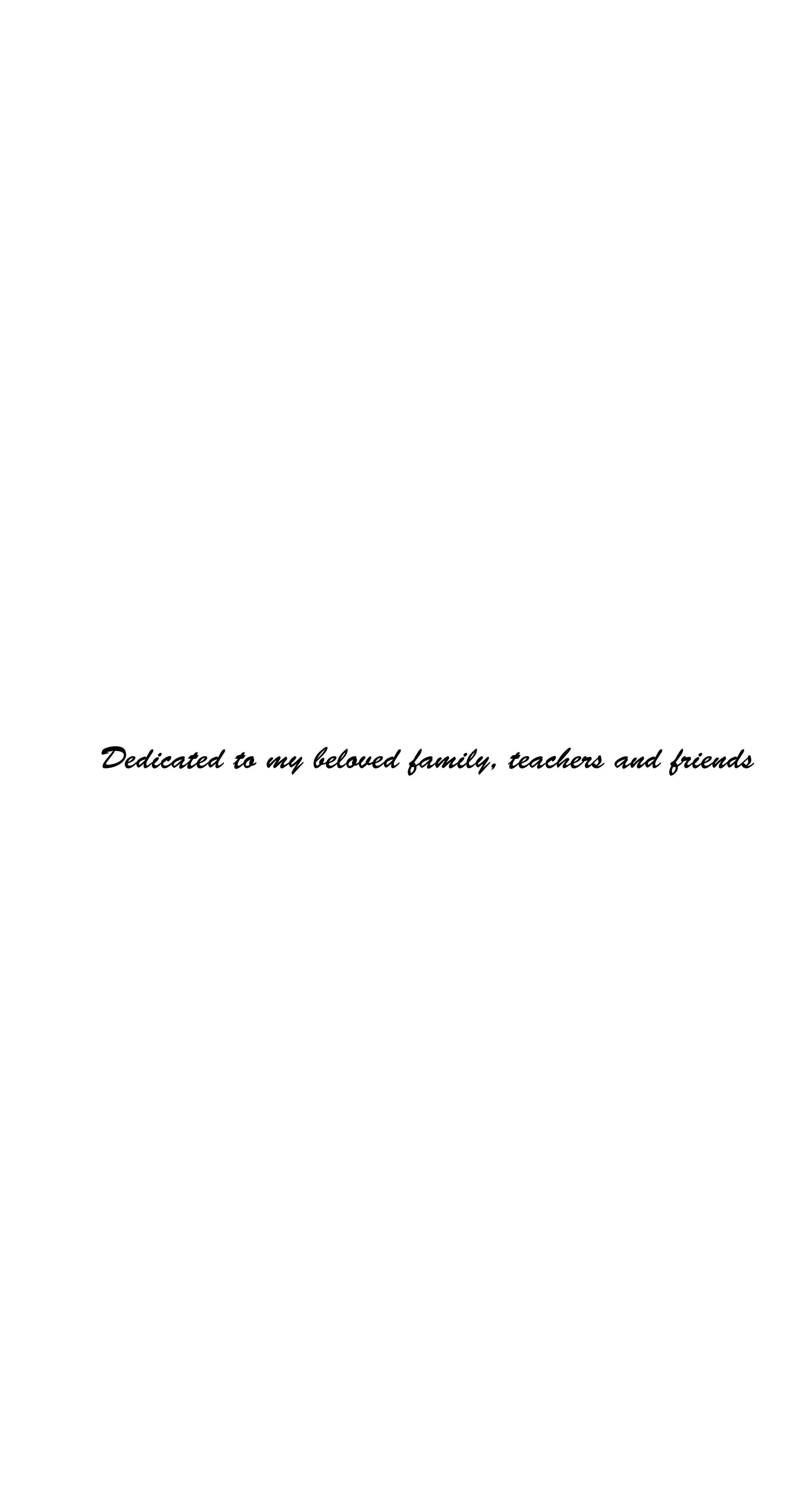
Chapters in Books and Lecture Notes

(a) "Distinguishing Attacks on (Ultra-)Lightweight WG Ciphers", Joseph, M., Sekar, G., Balasubramanian, R., *Proceedings of LightSec 2016, LNCS*, **2017**, *10098*, 45–59.

Conferences

(a) "Fault-Assisted Side Channel Analysis of HMAC-Streebog", Joseph, M., Sekar, G., Balasubramanian, R., *Preproceedings of CTCrypt 2020*, **2020**, 186–201.

Mabin Joseph Puthiakulangara



Dedicated to my beloved family, teachers and friends

Acknowledgements

No one who achieves success does so without the help of others. The wise and confident acknowledge this help with gratitude.

– Alfred North Whitehead

Now I am close to the culmination of an exciting journey — the journey I started pursuing my dreams. I am forever indebted to the Almighty for bringing me this far. At this moment, I want to extend heartfelt gratitude to all the wonderful people around me whose help and motivation made my journey to PhD successful and memorable.

First and foremost, I thank my Guide, Prof. Dr. R. Balasubramanian (whom we affectionately call Balu sir), and my Technology Advisor, Dr. Gautham Sekar, for directing me along the right path. I am so grateful to Balu sir for accepting my request to mentor me and allowing me to work with him. Through his life, words and actions, he taught me that '*with humility comes wisdom*'. His technical and personal support immensely helped me in various stages of my doctoral research. I am much obliged to Dr. Gautham Sekar for steering me through this journey with his profound expertise in symmetric-key cryptology. The first time when I approached him and expressed my desire to learn symmetric-key cryptanalysis, I could hardly foresee how much I was to mature and gain intellectually in the years to come. Nevertheless, the overwhelming support he provided by guiding me to identify research problems, reviewing my results and proofreading my papers — even during the period when he was not officially part of

my doctoral committee — helped me to reach this level. I have been extremely fortunate to associate with him.

I am honoured to have Dr. Amarendra G., Dr. B. P. C. Rao, Prof. Dr. C. Pandu Rangan, Dr. B. K. Panigrahi, Dr. Sharat Chandra, Dr. V. S. Srinivasan and Dr. S. A. V. Satya Murty serve as my doctoral committee members. Their critical assessment of my work and insightful feedback motivated me to explore deeper into my research domain. I sincerely thank Dr. S. A. V Satya Murty for introducing the field of cryptology to me and giving me the opportunity to pursue research in symmetric-key cryptology. I also thank Dr. Anish Kumar, Dean (Engg Sciences) and Dr. G. Sasikala, Former Dean (Engg Sciences), for their support on various administrative formalities. I also benefited a lot from the courses taught by Prof. V. Arvind and Prof. Venkatesh Raman, and the technical discussions with Prof. Santanu Sarkar and Dr. Prem Laxman Das.

Many thanks to all the coauthors of my papers; those not thanked above include Dr. G. Venkiteswaran. I would also like to thank the anonymous referees of my papers for their constructive comments.

I am extremely grateful to my immediate superiors at Computer Division (IGCAR), Mr. R. Jehadeesan, Mr. J. Rajan and Mr. K. Vijaykumar, for their wholehearted support. I thank my friend and colleague, Mr. Prasanth Kumar Thandra, for helping me build the fundamentals of cryptology and providing valuable insights into my doctoral study. I extend my sincere gratitude to Mr. Mohit Kumar Yadav for sharing my professional workload; his generosity helped me spend more time on my research. Many thanks to Mr. Prasanth, Mr. Sreejith and Mr. Shalabh for giving me good company at the office. I also take this opportunity to thank all my friends at IGCAR for their support and encouragement.

It is a great delight for me to remember my teachers, friends and relatives at this special moment. Since there are too many of them, I am sorry at my inability to provide names here. Their love, support, encouragement, motivation and prayers helped me to move forward, even during tough times.

Finally, I want to express my greatest gratitude to my family. This journey would have been impossible without the love and blessings of my Parents. I am at a loss for words to thank Pappa and Mummy for being the best part of my life. I am so fortunate to have my better half, Neena — her continuous support, love and affection made my life easier — and my little stars, Johann and Mikhail; they are my everything. I am also thankful to my sister Mebil, uncle Fr. Hans and aunt Sr. Jyothi, for their love and concern throughout the years.

Mabin Joseph Puthiakulangara

Kalpakkam, July 2022

Summary

Symmetric-key cryptographic algorithms are the oldest form of cryptographic algorithms to secure communication between two parties having identical keys. There are three types of modern symmetric-key algorithms — stream ciphers, block ciphers and MAC algorithms. This thesis is devoted to the security analysis of some popular symmetric-key algorithms — including lightweight constructions — and their software implementations.

We begin with a discussion on some of the potential flaws in probability-based cryptanalysis. As a case study, the related-key distinguishing attacks on the stream ciphers Py, Pypy, TPy, TPypy, RCR-64 and RCR-32 proposed in a paper published in the *Journal of Universal Computer Science* are reviewed. We show that the computations that led to the alleged attacks are flawed and establish the non-existence of the keystream biases detected in the Py family of stream ciphers.

Following this, we present distinguishing attacks on the Welch-Gong (WG) family of stream ciphers. The WG family of stream ciphers include two subfamilies, which we call WG-A and WG-B, of patented (ultra-)lightweight ciphers designed by Gong et al. The Waterloo Commercialization Office, Canada, has included the WG-A in an RFID anti-counterfeiting system and has proposed the WG-B for securing 4G/5G networks. Our attacks exploit the input-output correlations in the nonlinear transformations used by these ciphers. These are the first attacks on these ciphers.

Next, we show carry flag attacks on the unprotected implementations of SPECK family of lightweight block ciphers and HMAC-Streebog family of MAC algorithms.

SPECK is an ISO/IEC standard for RFID devices developed by Beaulieu et al. of the NSA. HMAC–Streebog is a MAC algorithm based on the Streebog, a family of hash functions defined in the Russian cryptographic standard GOST R 34.11–2012. These symmetric-key algorithms use modular addition, making them vulnerable to carry flag attacks. To the best of our knowledge, this thesis presents the first results analysing the resistance of unprotected implementations of the SPECK and the HMAC-Streebog to carry flag attacks. Our attacks, which work on the full SPECK, are comparatively more feasible than the other attacks applicable on the full ciphers. We present two types of side-channel attacks on the HMAC-Streebog: passive attacks without fault injections and active attacks with fault injections. Our passive attack is the best non-fault attack on HMAC-Streebog-256. Similarly, our active attacks fare better than the existing fault attacks on HMAC–Streebog as they have a larger temporal window for fault injection, target a more accessible location and cannot be mitigated with output redundancy countermeasures.

In the final part of this thesis, we analyse the RCR family of stream ciphers and their software implementations. The RCR ciphers have remained unbroken since they were published in 2007. We present arguments that not only support the designers' security claims but suggest, in general, that the ciphers are secure against several classes of cryptanalytic attacks. We also suggest ways to protect software implementations of the RCR ciphers against (cache-)timing and processor flag attacks. Our performance evaluation suggests that the protected implementation of the RCR-64 encrypts long messages at speeds comparable to some of the fastest stream ciphers available today. This is the first work to present a detailed study on the security and performance of the RCR ciphers.

Contents

Recommendations of the Viva Voce Committee i

Statement by Author ii

Declaration iii

List of Publications iv

Acknowledgements vi

Summary ix

Contents xi

List of Figures xv

List of Tables xviii

List of Symbols xxii

List of Abbreviations xxiv

1 Introduction 1

1.1 Cryptography 1

1.1.1 Cryptographic Primitives 2

1.1.2 Public-key Ciphers and Digital Signature Schemes 2

1.1.3 Cryptographic Hash Functions 3
1.2 Symmetric-key Primitives 4
1.2.1 Stream Ciphers 5
1.2.2 Block Ciphers 9
1.2.3 MAC Algorithms 16
1.2.4 Authenticated Encryption Algorithms 18
1.3 Shannon's Theory of Secrecy 21
1.3.1 Perfect Secrecy 21
1.3.2 Entropy and Redundancy 22
1.3.3 Spurious Keys and Unicity Distance 23
1.4 Importance of the Secret Key 24
1.5 Lightweight Cryptography 25
1.5.1 Lightweight Stream Ciphers 26
1.5.2 Lightweight Block Ciphers 28
1.6 Cryptanalysis 30
1.6.1 Algorithmic Cryptanalysis 32
1.6.2 Side-Channel Analysis 35
1.6.3 Fault Analysis 38
1.7 Motivation, Objective and Scope 39
1.8 Outline of the Thesis 40

2 On the Security of the Stream Ciphers RCR-64 and RCR-32 44

2.1 Introduction 44
2.2 Probability Assumptions and Computations in Cryptanalysis 47
2.2.1 Independent Events 47
2.2.2 Pairwise Disjoint Events 47
2.2.3 High Probability Events 48
2.2.4 Uniformly Distributed Random Variables 48
2.3 The Alleged related-key Distinguishing Attacks by Ding et al. 49

2.3.1 Specifications of the Py family of Stream Ciphers 49
2.3.2 Description of Ding et al.'s Alleged Related-Key Attacks 49
2.3.3 Observations on Ding et al.'s Alleged Attacks 57
2.4 Conclusions 61

3 Distinguishing Attacks on (Ultra-)lightweight WG Ciphers 62
3.1 Introduction 62
3.2 Specifications of the Ciphers 65
3.2.1 WG-A 65
3.2.2 WG-B 67
3.3 Motivational Observation 68
3.4 Bias Estimation 72
3.4.1 Biases in the Keystream of WG-A_{61} 72
3.4.2 Biases in the Keystream of WG-B_{157} 74
3.4.3 Improvements to the Bias Estimations 74
3.5 Attack Complexities 75
3.5.1 Experimental Verification 76
3.6 Discussion 77
3.7 Conclusions 78

4 Side-Channel Analysis of SPECK 79
4.1 Introduction 79
4.2 Specifications of the Ciphers 85
4.2.1 $\text{SPECK}_{m,n}$ 85
4.3 Motivational Observation 88
4.4 Key Recovery 90
4.4.1 Experimental Verification 93
4.4.2 Reducing the Time Complexity of the Attack 95
4.5 Attack on Key Schedule 97

4.6 Discussion 99
4.7 Conclusions 105

5 Fault-Assisted Side-Channel Analysis of HMAC-Streebog 107

5.1 Introduction 107
5.2 Specification of HMAC-Streebog 109
5.2.1 Description of Streebog 109
5.2.2 Description of HMAC-Streebog 111
5.3 Motivational Observations 112
5.3.1 On Carry in Modular Addition 112
5.3.2 On the Recovery of an Unknown Operand 114
5.3.3 Passive Analysis vs Active Analysis 116
5.4 Key Recovery Attacks on HMAC-Streebog 118
5.4.1 Complexities of the Attacks 119
5.5 Validity of the Assumptions 120
5.5.1 Extraction of the Inner Hash H_{in} 121
5.5.2 Detection of Carry 121
5.5.3 Chosen Bit Modification of Inner Hash H_{in} 122
5.6 Conclusions 123

6 Revisiting the Software-Efficient Stream Ciphers RCR-64 and RCR-32 124

6.1 Introduction 124
6.2 Specifications of the Ciphers 127
6.2.1 RCR-64 and RCR-32 127
6.2.2 Recommended Key and IV Sizes 127
6.3 Statistical Analysis 130
6.3.1 Keystream analysis 130
6.3.2 Structural analysis 133
6.4 Security Analysis 136

6.4.1 Resistance to differential cryptanalysis 136
6.4.2 Resistance to linear cryptanalysis 138
6.4.3 Resistance to algebraic cryptanalysis 139
6.4.4 Resistance to cube attacks 139
6.4.5 Resistance to time-memory-data trade-off (TMDTO) attacks . . 140
6.4.6 Resistance to distinguishing attacks 142
6.4.7 Resistance to (cache-)timing attacks 144
6.4.8 Resistance to carry flag attacks 145
6.5 Performance Evaluation 146
6.6 Conclusions 149

7 Closing Remarks 150
7.1 Conclusions 150
7.2 Future Research 152

Bibliography 155

List of Figures

1.1 The ECB mode of encryption 12

1.2 An example showing how the ECB mode of encryption reveals the patterns in an image (first bitmap image was encrypted using AES-128 in ECB mode to generate the second image) 12

1.3 The CBC mode of encryption 13

1.4 The CFB mode of encryption 14

1.5 The OFB mode of encryption 14

1.6 The counter mode of encryption 15

1.7 Encrypt-then-MAC (EtM) approach to combine encryption and MAC algorithms 19

1.8 Encrypt-and-MAC (E&M) approach to combine encryption and MAC algorithms 20

1.9 MAC-then-Encrypt (MtE) approach to combine encryption and MAC algorithms 20

1.10 An LFSR of length l whose taps and stages (at time t) are represented by c_i and s_{t+i}, respectively, where $l-1 \leq i \leq 0$ 27

2.1 Visual representation of the round function of the Py family of ciphers, where (Y^j, P^j, s^j) and Z^j are the internal state at the beginning of jth round and the output word generated from it, respectively 53

3.1 Keystream generation of WG-A$_d$; exp computes $(s_{\text{A}}[19])^d$ 66

3.2 Keystream generation of WG-B$_d$; exp computes $(s_{\text{B}}[31])^d$ 68

4.1 The $(i+1)$th round of SPECK$_{m,n}$ 86

4.2 Final round of encryption of SPECK$_{32,64}$ 88

4.3 Program to recover the 10 most significant bits of the final subkey of SPECK$_{32,64}$ 96

4.4 Final round of the key expansion phase of SPECK$_{32,64}$ 98

4.5 Simulation of the SPECK implementation for AVR microcontrollers available in the FELICS project [211] to confirm that the outgoing carry at the MSB position of modular addition in the final round agrees with the carry flag at the end of encryption, if the left circular shift operation of the final round is skipped 101

4.6 Last two rounds of encryption of SPECK$_{32,64}$ 104

5.1 Probability distributions of the size of Algorithm 15's solution space for n = 4, 8, 12 and 16 116

5.2 Minimum number of simultaneous bit-flips (l) required to recover the k most significant bits for each of the success probabilities 0.6, 0.7, 0.8, 0.9 and 0.95 117

6.1 Visual representation of the round functions of the RCR ciphers, where (Y^j, P^j, s^j) and Z^j are the internal state at the beginning of jth round and the output word generated from it, respectively 131

6.2 Propagation of the XOR difference $\triangle_{s,0} := \delta_{j,0}$ through the loops A and B of Algorithm 17 to ($\triangle_{s,t+ivb+1}$, $\triangle_{EIV}$), where $\delta_{j,0}$ is the difference at the $(j+1)$th byte of the IV, ivb is the IV size in bytes, $\triangle_{s,i+1}$ and $\delta_{j,i+1}$ are the differences in s and the output of $P[\cdot]$, respectively, after the $(i+1)$th step ($0 \leq i \leq t+ivb+1$; $t = ivb-1-j$) and, $\triangle_{EIV}$ and $\triangle_{EIV,i}$ are the differences in EIV and its $(i+1)$th byte ($0 \leq i \leq ivb-1$), respectively 138

6.3 Encryption timings (in CPU cylces) of unprotected and protected implementations of the RCR ciphers for all message lengths between 1 and 4096 bytes 147

List of Tables

2.1 The performances of a few stream ciphers, including the AES in counter mode, to encrypt 1536-byte, 4096-byte and much longer messages with 256-bit keys on Intel Core i5-1030NG7 processor as measured by the eBASC project 46

2.2 Notation for different parts of the ciphers Py, Pypy, TPy, TPypy, RCR-64 and RCR-32 50

2.3 The probabilities p_1, p_2, p_3 and p_4, where $p_1 = \Pr(t_3 = 0 \mid X)$, $p_2 = \Pr(t_4 = 0 \mid X)$, $p_3 = \Pr(t_1 \oplus t_2 = 0 \mid X)$ and $p_4 = \Pr(\widehat{Z} = 0 \mid X)$; X represents different combinations of the events D, G and E 59

2.4 The probabilities p_1, p_2, p_3 and p_4, where $p_1 = \Pr(t_3 = 0 \mid Y)$, $p_2 = \Pr(t_4 = 0 \mid Y)$, $p_3 = \Pr(t_5 \oplus t_6 = 0 \mid Y)$ and $p_4 = \Pr(\widehat{Z} = 0 \mid Y)$; Y represents different combinations of the events D, G and F 60

3.1 Attacks on the WG family of stream ciphers 64

3.2 WG: Probabilities that (a) $W_{\text{A}}(x^{61}) = x_{(i)}$ and (b) $W_{\text{A}}(x^{61}) = (x \boxdot_8 \omega^{38})_{(i)}$, for several values of i 69

3.3 WG: Probabilities that (a) $W_{\text{B}}(x^{157}) = x_{(i)}$ and (b) $W_{\text{B}}(x^{157}) = (x \boxdot_{16} (\beta^2 + 1))_{(i)}$, for several values of i 69

3.4 Truth table that satisfies the relation between the Boolean variables Y_1, Y_2, Y_3, Y_4 and Y_5 73

3.5 Probabilities that (a) $\hat{z}_{\text{A}} = 0$ and (b) $\hat{z}_{\text{B}} = 0$, for each of the WG-A$_d$ and WG-B$_d$ ciphers . . . 75

3.6 Data requirements of our attacks (corresponding to 0.9999 success probability) on the WG-A$_d$ and WG-B$_d$ ciphers . . . 77

4.1 Comparative analysis of the best-known key recovery attacks (with a success rate of 99.99%) on the SPECK ciphers in chronological order . 82

4.2 Parameters of SPECK$_{m,n}$ . . . 86

4.3 Truth table showing the allowed values of the Boolean variables $x_{(i)}, y_{(i)}$, c_i, $z_{(i)}$ and c_{i+1} . . . 90

4.4 Probabilities that $z_{(i)} \oplus c_{out} = 1$, for $15 \geq i \geq 0$, in SPECK$_{32,64}$ . . . 91

4.5 Data requirements for SPECK$_{32,64}$ (corresponding to 0.9999 success probability) to determine if b_i is biased to 1 or 0, for $15 \geq i \geq 0$ . . . 92

4.6 Values of $p_i - 0.5$, t_i and the success probabilities ω_i corresponding to $N = 2^{30}$, for $15 \geq i \geq 0$ . . . 93

4.7 Success rates of our experiments on SPECK$_{32,64}$ to recover $k_{21(i)}$ with 2^{23} samples, for $15 \geq i \geq 0$ . . . 95

4.9 Complexities of the full key recovery attacks on SPECK$_{m,n}$ corresponding to the success rates (SR) of 99.97%, 98.48% and 84.91%, under the assumption that the carry flag at the end of the key expansion phase is known . . . 99

4.10 Pseudocodes of the final rounds of the two software implementations of SPECK, where r is the number of rounds and (x, y) are the inputs to the final round . . . 100

4.11 Vulnerability of SPECK$_{m,n}$ when implemented on ℓ-bit microcontroller, where $\ell = 8, 16$ or 32 . . . 102

5.1 Table showing the allowed values of the Boolean variables $x_{(i-1)}$, $y_{(i-1)}$, $c_{(i-1)}$, $c_{(i)}$ and $c'_{(i)}$, and the probability of each event where p_i = Pr($c_{(i)}$ = 0), for $i > 1$ . . . 112

5.2 Variation of the number of key bits recovered ($i \cdot l$) using Algorithm 15 and the required number of modifications of H_{in} in logarithmic scale ($\log_2 \tau_{i \cdot l}$) with the possible number of chosen bit-flips (l) where i = 1.4, 1.48, 1.57, 1.6, 1.67 and $i \cdot l \leq 512$ . . . 119

6.1 The results of the NIST statistical tests on the keystreams generated by the RCR ciphers, where p is the proportion of the sequences that passed the test . . . 132

6.2 The mean P-value computed for each structural analysis test on RCR-64 and RCR-32 . . . 136

6.3 The performances of a few stream ciphers, including the AES, SPECK and SIMON in counter mode, to encrypt 64-byte, 576-byte, 1536-byte, 4096-byte and much longer messages with 256-bit keys on Intel Core i5-680 processor measured using SUPERCOP toolkit . . . 148

List of Symbols

Operators

$\lll$	Bitwise circular left shift.
$\ggg$	Bitwise circular right shift.
$\gg$	Bitwise right shift.
$\oplus$	Bitwise exclusive OR.
&	Bitwise AND.
$\boxplus_n$	Addition in $\mathbb{F}_{2^n}$.
$\boxdot_n$	Multiplication in $\mathbb{F}_{2^n}$.
x^k, $x \in \mathbb{F}_{2^n}$	$\underbrace{x \boxdot_n x \boxdot_n x \boxdot_n \cdots \boxdot_n x}_{k}$.
$\boxplus_{(w)}$	Addition modulo 2^w where w is the word size in bits.
$\boxminus_{(w)}$	Subtraction modulo 2^w where w is the word size in bits.

Additional Notation and Conventions

$x_{(j)}$	$(j+1)$th bit of x ($j = 0$ denotes the least significant bit).
x'	One's / bitwise complement of x.
$\|x\|$	Length of x in bits.
$ROTL32(x, i)$	Cyclic left rotation of 32-bit x by i bits.
$x \parallel y$	Concatenation of the binary strings x and y.
$X[b]$	bth element of array X.
$\mathbb{F}_{2^n}$	Finite field of order 2^n.
$F_j(x)$	$(j+1)$th least significant byte of x.

$\Psi_j(x)$	$(j+1)$th least significant 64-bit word of x.
$I_{n,j}$	n-bit binary string such that $I_{n,j(i)} = 1 \iff i = j$.
$\{b\}^k$	$(k \cdot \lvert b \rvert)$-bit binary string $bbb \cdots b$.
$f_1 f_2(x)$	$f_1\left(f_2(x)\right)$, where f_1 and f_2 are functions.

List of Abbreviations

3G	third generation of wireless mobile telecommunications technology
AES	Advanced Encryption Standard
ALU	arithmetic logic unit
ARX	addition-rotation-exclusive-OR
CHF	cryptographic hash function
CP	chosen plaintext
CPU	central processing unit
DES	Data Encryption Standard
DRAM	dynamic random access memory
ECC	elliptic curve cryptography
FSM	finite state machine
GSM	Global System for Mobile communications
IEC	International Electrotechnical Commission
HMAC	hash-based message authentication code
IoT	Internet of Things
IPsec	Internet Protocol security
ISO	International Organization for Standardization
IV	initialization vector
KGA	keystream generation algorithm
KP	known plaintext
KSA	key scheduling algorithm

LFSR	linear feedback shift register
LSB	least significant bit
LWC	lightweight cryptography
MAC	message authentication code
MD	message-digest algorithm
MSB	most significant bit
NIST	National Institute of Standards and Technology, USA
NSA	the US National Security Agency
PKC	public-key cryptography
PRNG	pseudorandom number generator
RAM	random access memory
RFC	Request for Comments
RFID	radio frequency identification
ROM	read-only memory
RSA	Rivest-Shamir-Adleman algorithm
SPN	substitution / permutation network
SRAM	static random access memory
SSH	Secure Shell protocol
TLS	Transport Layer Security
UMTS	Universal Mobile Telecommunications System
XOR	exclusive-OR

Chapter 1

Introduction

1.1 Cryptography

Cryptography is the science that studies various mathematical aspects of information security used to protect data that is communicated over an insecure channel or stored in an unprotected medium. As Rivest said, it "is about communication in the presence of an adversary" [1]. Though historical records indicate the usage of secret messages around 4000 years ago, cryptography, as we see it today, is just less than 100 years old. The military and governments predominantly used classical cryptography to maintain the confidentiality of the information communicated. With the advent of computers and the internet, cryptography became more and more ubiquitous by being part of a multitude of applications, and now each of us knowingly or unknowingly employs it while using internet banking, e-commerce, smartphones, Internet of Things (IoT), etc.

Apart from confidentiality, which is considered the central aspect of information security, modern cryptography focuses on three fundamental services: integrity, authentication, and non-repudiation. The set of algorithms used to provide these information security functions is known as *cryptosystem*. A present-day cryptosystem addresses the following:

- Secrecy of data

- Unauthorised alteration of data
- Identification of the communicating parties
- Preventing one from denying his actions

1.1.1 Cryptographic Primitives

Cryptographic primitives are the fundamental cryptographic algorithms that are used as building blocks for higher-level cryptographic algorithms or security protocols. Let us consider an insecure channel over which two parties — Alice and Bob — are communicating and the adversary — Eve — is trying to eavesdrop on the communication. Alice and Bob use cryptographic primitives, which may or may not require a numerical parameter called the key, to protect their messages from Eve. Based on the usage of the key, they can be divided into the following three categories:

- Symmetric-key primitives that require Alice and Bob to share a secret key (more on symmetric-key primitives to follow in Sect. 1.2)
- Public-key primitives which need a public / private key pair to secure the communication in one direction
- Unkeyed primitives that do not require a key to implement the information security function

1.1.2 Public-key Ciphers and Digital Signature Schemes

These are the two popular types of cryptographic algorithms that constitute the public-key primitives. Cipher is an algorithm that performs the operations encryption and decryption to protect the confidentiality of the messages communicated between Alice and Bob. During encryption, Alice converts the message in its original form (plaintext) into an unintelligible form (ciphertext) using the encryption key K_e. On receiving the ciphertext, Bob performs the decryption operation to convert it back to the plaintext

using the decryption key K_d. For a public-key cipher, K_e (resp. K_d) forms the public (resp. private) key such that the message encrypted using the public key K_e can be decrypted only using the private key K_d. Similarly, a digital signature scheme enables Alice to confirm the authenticity of a message from Bob by verifying the signature on the message, which Bob generated with his private key K_d, using Bob's public key K_e. The focus of this thesis is not on public-key cryptography and there is no overlap between the works presented in this thesis and public-key cryptography. Therefore, the reader may refer to [2, 3, 4] for further details on popular public-key cryptosystems such as RSA and ECC.

1.1.3 Cryptographic Hash Functions

These are the classic examples of unkeyed cryptographic primitives which are used to verify the integrity of the data. A function $h(\cdot)$ whose domain is larger than its range must have the following properties to qualify as a cryptographically secure hash function.

1: Collision resistance: It should be computationally infeasible to find two distinct points x_1 and x_2 in the domain of $h(\cdot)$ such that $h(x_1) = h(x_2)$. The upper bound of the collision resistance of an n-bit hash function is set as $2^{n/2}$ by the birthday attack [5].

2: First preimage resistance (often known as preimage resistance): For any y in the range of the n-bit hash function $h(\cdot)$, it should not be possible to find x with fewer than 2^n evaluations of $h(\cdot)$ such that $h(x) = y$.

3: Second preimage resistance: For a given x in the domain of the $h(\cdot)$, it should be computationally infeasible to find another point x' in the domain such that $h(x) = h(x')$. For an n-bit hash function, second preimage resistance has an upper bound of 2^n.

In addition to verification of message integrity, CHFs are mainly used in MAC algorithms, in digital signature schemes, as PRNGs, etc. Many of the popular CHFs such as MD5 [6], SHA-1 [7], SHA-2 [8], Whirlpool [9], Streebog [10], etc. are based on a design technique known as the Merkle-Damågard construction, which was independently devised by Merkle [11] and Damågard [12]. It is an iterative construction scheme that is used to build collision-resistant CHFs from collision-resistant one-way compression functions. The compression function takes a message block of fixed size (known as the block size of the CHF) as input and outputs a shorter block. A CHF based on Merkle-Damågard construction computes the hash of a message $M := M_0 \parallel M_1 \parallel \ldots M_{n-1}$ as follows:

$$H_{i+1} = f(H_i, M_i)\ ,\ i = 0, 1, \ldots, n-1\ ,$$

where $f(\cdot)$ is the compression function, H_0 is the initialization vector, $|M_i|$ is the block size of the CHF and H_n is the resultant hash value. To process arbitrarily long messages and also to enhance the security, the final message block is often padded with bits that encode the length of the message.

Any CHF discussed in this thesis could be assumed to be based on the Merkle-Damågard construction unless explicitly mentioned.

1.2 Symmetric-key Primitives

The three most popular symmetric-key primitives are the following:

- Stream ciphers
- Block ciphers
- MAC algorithms

As the term "ciphers" suggests, stream ciphers and block ciphers — collectively known as symmetric-key ciphers — are algorithms used to protect the confidentiality of

data in a symmetric-key setting. Until 1976, when Diffie and Hellman proposed the first public-key cryptosystem [13], symmetric-key ciphers were the only known algorithms for encryption. Compared to public-key ciphers, which rely on complex mathematical problems for security, symmetric-key ciphers are constructed using simple operations which can be efficiently implemented on modern computers. Therefore, stream ciphers and block ciphers are many-folds faster than their public-key counterparts. Due to this reason, security protocols generally use symmetric-key ciphers, whose keys are securely exchanged using public-key encryption algorithms, to encrypt the actual data. Thus symmetric-key algorithms are considered as the workhorses of the cryptographic world.

This thesis will be primarily focusing on the study of stream ciphers, block ciphers and MAC algorithms. The fundamental aspects of each of these primitives will be discussed in the following subsections.

1.2.1 Stream Ciphers

Stream ciphers are the class of encryption algorithms which encrypt each bit of a plaintext message one at a time. They are well suited for applications where streaming data has to be processed without any delay. One should be aware of the one-time pad before knowing the stream ciphers. For an n-bit plaintext message m, the one-time pad generates an n-bit ciphertext c using the operation:

$$c_{(i)} = m_{(i)} \oplus k_{(i)} ,$$

where k is an n-bit key whose bits are generated independently and randomly. According to Shannon, in order to achieve unconditional security, the secret key must be at least as uncertain as the plaintext [14]. Despite the fact that the one-time pad has the smallest possible key to satisfy Shannon's condition for unconditional security, it cannot be part of a practical cryptosystem as it is difficult to distribute and manage a key which is as long as the plaintext. In order to overcome this problem, stream ciphers were introduced.

A stream cipher is a cryptographic algorithm that takes as input the secret key, which is much shorter than the plaintext, and a parameter called the initialization vector (IV), and outputs a pseudorandom keystream sequence as long as the plaintext. Similar to the one-time pad, the stream cipher generates the ciphertext by mixing the plaintext with the keystream using the XOR operation. Since the keystream is a pseudorandom sequence, the stream cipher does not satisfy Shannon's condition for unconditional security. Nevertheless, under the assumption that the keystream "appears random" to a computationally bounded adversary, it is considered to be computationally secure.

For a cipher to be semantically secure, the ciphertext should not leak any information about the plaintext. In the absence of an IV, stream ciphers can achieve semantic security only if they avoid reusing the key to encrypt distinct messages. If the same key is used to encrypt two partly identical messages, the resulting ciphertexts will also have a pattern similar to the plaintexts. If the messages are wholly distinct, reusing the key will result in generating ciphertext pairs whose XOR sum will be equal to that of the corresponding plaintext pairs as discussed in the transmission in depth problem [15]. Since frequent rekeying is not always possible, the IV, which is used only once with a key, is the indispensable parameter that makes a stream cipher semantically secure. Depending on the application, initialization vectors may be generated incrementally or randomly, and are often publicly known.

In general, the following three algorithms constitute a stream cipher.

- *The Key Scheduling Algorithm (KSA).* This algorithm takes the secret key K as input and generates the internal state of the stream cipher.
- *The IV Scheduling Algorithm (IVSA).* The internal state generated by the KSA is updated using this algorithm by mixing the IV with it.
- *The Keystream Generation Algorithm (KGA).* The internal state initialised using the KSA and IVSA — collectively known as the key/IV setup — is given as input to this algorithm. During each iteration, KGA generates a fixed-length keystream

and updates the internal state. Though it is possible to generate an indefinitely long keystream sequence from a key/IV pair, most of the stream ciphers will have a restriction on its length for enhanced security.

Stream ciphers can be further categorised into two types: synchronous stream ciphers and asynchronous stream ciphers. Let us discuss them in some detail.

Synchronous Stream Ciphers. The stream ciphers that generate keystreams independently of the plaintext and ciphertext belong to this category. Let f be the function that implements the KGA, z_i and s_{i+1} be the keystream block and the internal state, respectively, generated at the end of ith round of the KGA, where $i = 0$ indicates the first round, and s_0 be the internal state at the end of the key/IV setup. A synchronous stream cipher can be represented in one of the following ways depending on whether the key or the IV is reused in the KGA:

$$(z_i, s_{i+1}) = f(K, IV, s_i)\,,$$

$$(z_i, s_{i+1}) = f(K, s_i)\,,$$

$$(z_i, s_{i+1}) = f(IV, s_i)\,,$$

$$(z_i, s_{i+1}) = f(s_i)\,.$$

Synchronous stream ciphers have the following properties:

1: Since they cannot self-synchronise, the sender and the receiver must employ additional mechanisms to synchronise the keystream generation.

2: Flipping of a ciphertext bit due to transmission error will not affect the decryption of the remaining bits.

3: Insertion or deletion of a ciphertext bit by an active adversary or due to transmission error will affect the decryption of the subsequent bits until the synchronisation is regained.

Synchronous stream ciphers, which are often constructed using linear feedback shift registers (LFSRs) or arrays, are more popular and well studied than asynchronous stream ciphers. Both LFSR-based and array-based synchronous stream ciphers have been analysed in this thesis. LFSR-based constructions are usually hardware-efficient, and the most widely used among them include E0 (used in the Bluetooth standard) [16], A5/1 (used in the GSM cellular telephone standard) [17], SNOW 2.0 (an ISO/IEC stream cipher standard) [18, 19] and its variant SNOW 3G (used in UMTS 3G networks) [20]. When compared to LFSR-based ciphers, array-based stream ciphers are considered to be best-suited for software applications. The stream cipher RC4, designed by Rivest in 1987, is the first and most popular array-based stream cipher [21, 22]. Being remarkably simple and fast in software, RC4 has inspired the designs of many array-based ciphers such as the HC and Py families of stream ciphers [23, 24, 25].

Asynchronous Stream Ciphers. The stream ciphers that generate the keystream block as a function of the key and a fixed number (say l) of previous ciphertext blocks are known as asynchronous or self-synchronising stream ciphers. They can be represented as:

$$z_i = f(K, c'_{i-1})\,,$$

where c'_{i-1} represents the l ciphertext blocks $c_{i-l-2}, c_{i-l-1}, \ldots, c_{i-1}$. The l dummy ciphertext blocks required to generate the keystream blocks $z_0, z_1, \ldots, z_{l-1}$ are derived from an initialization vector. The stream ciphers Helix, Phelix, SSS and MOUSTIQUE are well-known examples of asynchronous stream ciphers [26, 27, 28, 29]. These ciphers have the following properties:

1: They can self-synchronise even when some of the ciphertext bits get deleted or inserted during transmission.

2: Single-bit error in a ciphertext block can affect l plaintext blocks at the receiver's

end. In order to reduce the intensity of this problem, many asynchronous stream ciphers are designed with small l.

Henceforth, the term “stream cipher” will refer to synchronous stream ciphers unless explicitly mentioned.

Ideal Stream Cipher. A stream cipher whose keystream bits are distributed uniformly at random is considered to be an ideal stream cipher, i.e., if z is the keystream generated, for an ideal stream cipher, $\Pr(z_{(i)}) = 0.5$ for all $i \geq 0$. The most important design objective of a good stream cipher is to achieve this ideal behaviour.

1.2.2 Block Ciphers

A block cipher is an encryption function that maps an m-bit plaintext block to an m-bit ciphertext block using a secret key K. Being a one-to-one function, the inverse of it is used to decrypt the ciphertext block. In other words, for a fixed K the block cipher is a bijection that permutes m-bit vectors whose domain and range are the plaintext space and the ciphertext space, respectively. Block cipher can be considered as a universal symmetric-key primitive as it can be used to build both stream ciphers and message authentication codes, which we will see in the later part of the Section. Moreover, the constructions of Davies-Meyer [31], Matyas-Meyer-Oseas [32] and Miyaguchi-Preneel [33, 34] compression functions, which are used in CHFs, are also based on block ciphers.

Block cipher designs are based on the following two concepts introduced by Shannon in his landmark paper *Communication Theory of Secrecy Systems*: confusion and diffusion [14]. To create confusion, the relation between the ciphertext statistics and the plaintext statistics must be too complex for an attacker to exploit. Similarly, if each bit of the plaintext and the key can influence many bits of the ciphertext then we say that the cipher has good diffusion property. Most of the block ciphers achieve these two properties by using some combination of the two basic operations: substitution and permutation. The block cipher AES [30], which is a *de facto* standard

for encryption today, is designed solely using these operations. Such constructions, known as the Substitution-Permutation Networks (SPNs), encrypt a plaintext block by repeatedly performing certain carefully chosen substitutions and permutations along with the addition of the key material. Let us briefly look into the details of yet another construction called the Feistel Networks (FNs) based on which the block ciphers analysed in Chapter 4 has been designed.

Feistel Network. Though Shannon proposed the concept of a *product cipher*, which is regarded as the predecessor of modern block ciphers, in 1948 [14], it took more than a decade to start open research on block ciphers. The research conducted by the IBM Corporation under the supervision of Horst Feistel, culminated in the design of the Lucifer family of block ciphers, which are reckoned the first known construction based on FN [35]. Lucifer inspired the design of the US encryption standard DES [36], which was the most popular and widely deployed symmetric-key cipher until the introduction of AES.

An FN-based cipher or Feistel cipher can be represented as follows:

$$\begin{aligned} x_{i+1} &= y_i \,, \\ y_{i+1} &= x_i \oplus f(y_i, k_i) \,, \; i = 0, 1, \ldots, r-1 \,, \end{aligned}$$

where $P := x_0 \parallel y_0$ and $C := x_r \parallel y_r$ are the plaintext and ciphertext blocks, respectively, k_i is the subkey used in the $(i+1)$th round, $f(\cdot)$ is a time-invariant function often called the *round function* and r is the total number of rounds. The subkeys used in each round are generated from K using a KSA. In [37], Luby and Rackoff analysed the FNs and proved that given $f(\cdot)$ is a truly random function, the Feistel cipher will be a truly random permutation which is secure against chosen plaintext attacks, if $r \geq 3$. In order to make the cipher secure against both chosen plaintext and known plaintext attacks, the number of rounds has to be greater than 4 (more about CP and KP attacks

will be discussed in Sect. 1.6). Therefore, if $f(\cdot)$ and the KSA are strong, a small r is sufficient for a secure Feistel cipher. Some of the notable examples of Feistel ciphers, other than DES and Lucifer, include Blowfish [38], Camellia [39], KASUMI [40], GOST [41] and TDEA [42].

As we have already seen, block ciphers are inherently designed to encrypt m-bit messages. A shorter message has to be "padded" with extra bits to make its length equal to the block size. In order to encrypt the messages that are longer than m bits, block ciphers are used in certain modes of operation. Let p be a message of size $t \cdot m$ bits which is divided into t blocks $p_0, p_1, \ldots, p_{t-1}$. Let E be a block cipher which encrypts an m-bit plaintext block using the key K to generate the ciphertext block c_i, for $0 \leq i < t$. The block cipher E can be typically used in one of the following five modes of operation, as specified in FIPS 81 [43] and NIST SP800-38A [44], to encrypt p. Since it is fairly straightforward to understand the decryption using these modes, their details will be omitted.

Electronic Code Book (ECB). As illustrated in Figure 1.1, in this mode of operation, the plaintext blocks are encrypted independently of one another. Since the ciphertext blocks corresponding to identical plaintext blocks in the message always remain the same, a block cipher used in ECB mode does not hide the patterns in the message as shown in Figure 1.2. Another disadvantage with the ECB mode is that shifting the positions of the plaintext blocks will get replicated in the ciphertext. Due to these reasons, ECB mode, which does not provide semantic security, is seldom used.

Cipher-Block Chaining (CBC). In this mode of encryption, a plaintext block is XORed with the previous ciphertext block before being given as input to the block cipher. A schematic representation of the CBC mode is shown in Figure 1.3. Similar to stream ciphers, an IV, which will be XORed with the first plaintext block, is required to

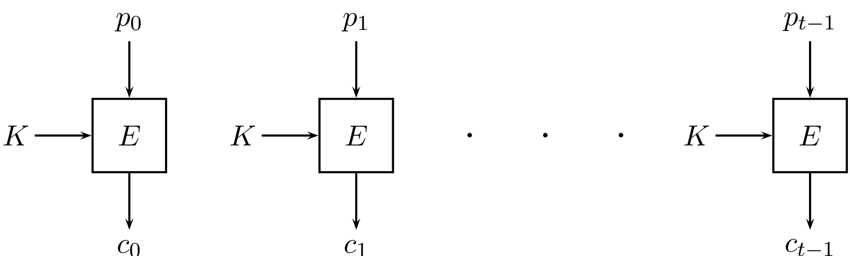


Figure 1.1: The ECB mode of encryption

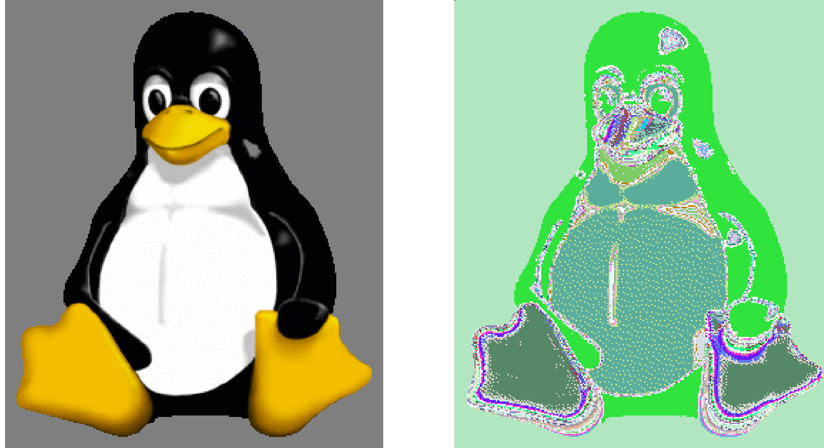


Figure 1.2: An example showing how the ECB mode of encryption reveals the patterns in an image (first bitmap image was encrypted using AES-128 in ECB mode to generate the second image)

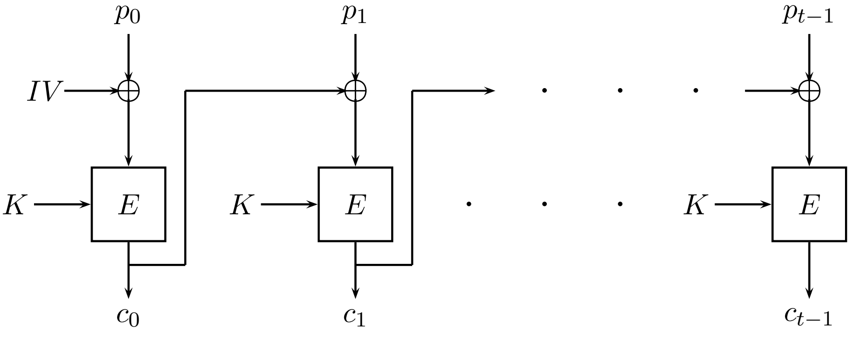


Figure 1.3: The CBC mode of encryption

initialise this mode of operation. Due to the chaining mechanism, each ciphertext block not only depends on the current plaintext block but also on all the preceding blocks. This benefits the encryption by providing semantic security. Despite that the chaining has two major disadvantages. Firstly, it limits the parallelisation of the encryption process. Secondly, it increases the number of blocks affected by the error propagation, which is one in ECB mode, to two.

Cipher Feedback (CFB). In this mode of encryption, the block cipher functions as an asynchronous stream cipher. The decryption of the messages encrypted using ECB and CBC modes of operation requires the function E^{-1}. Whereas, in CFB mode of operation E alone is enough to do both encryption and decryption. As shown in Figure 1.4, the first keystream block, which is the output of $E(IV, K)$ is XORed with the first plaintext block to generate the corresponding ciphertext block. The encryption of the subsequent plaintext blocks can be represented by:

$$c_i = p_i \oplus E(c_{i-1}, K)\ , \text{ for } 1 \leq i \leq t-1\ .$$

Since the ciphertext blocks are part of the chain, the advantages and disadvantages of the CBC mode are applicable to the CFB mode too.

Output Feedback (OFB). This mode of encryption, which is shown in Figure 1.5, can

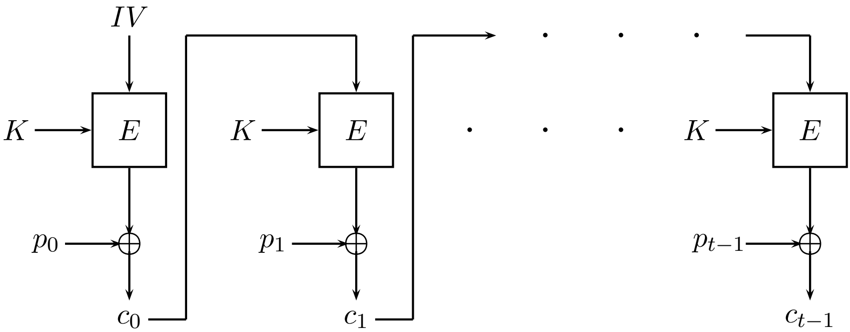


Figure 1.4: The CFB mode of encryption

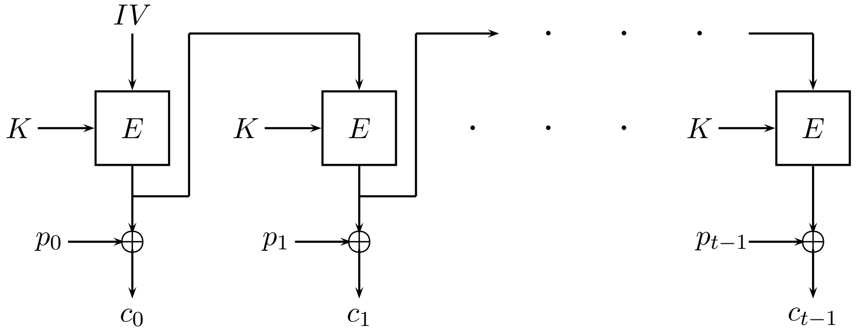


Figure 1.5: The OFB mode of encryption

be represented by:

$$z_i = E(z_{i-1}, K)\,,\ z_{-1} = IV\,,$$

$$c_i = p_i \oplus z_i\,,\ i = 0, 1, \ldots, t-1\,.$$

As in this mode the block cipher operates like a synchronous stream cipher, the error propagation is limited to one block.

Counter (CTR). Similar to the OFB mode, in this mode also, the block cipher is a synchronous stream cipher. It is represented by:

$$z_i = E(v_i, K)\,,$$

$$c_i = p_i \oplus z_i\,,\ i = 0, 1, \ldots, t-1\,,$$

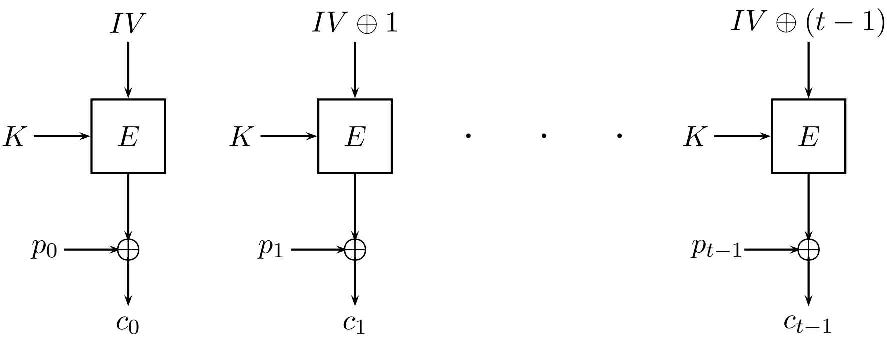


Figure 1.6: The counter mode of encryption

where the input to the block cipher v_i is generally generated from the IV and a counter j as $IV \parallel j$ or $IV \oplus j$ as shown in Figure 1.6. The CTR mode supports parallelisation due to the absence of the feedback mechanism and is therefore considered to be faster than the OFB mode.

The modes of operation discussed till now are meant to protect the confidentiality of the message. In practice, integrity and authenticity of the message also have to be protected and in symmetric-key setting MAC algorithms are used to achieve the same. Block ciphers can also be used in certain other modes of operation to perform *authenticated encryption* that provides confidentiality, integrity and authentication. Some of these modes are Offset Codebook (OCB) [45], Counter with CBC-MAC (CCM) [46], EAX Mode [47] and Galois/Counter (GCM) [48]; descriptions of these modes are beyond the scope of this thesis.

Ideal Block Cipher. We have seen that a block cipher will permute an m-bit input block using a k-bit secret key. The permutation varies depending on the key, and the total number of possible permutations equals $(2^m)!$, which can be approximated as $2^{(m-1)2^m}$ using Stirling's approximation [49]. For typical values of k and m, the block cipher will provide only a small fraction of the possible permutations. Therefore, a good block cipher is designed to behave like an ideal block cipher which will choose a permutation

uniformly at random from the $2^{(m-1)2^m}$ available choices, for a randomly chosen key.

1.2.3 MAC Algorithms

The main objective of the symmetric-key algorithms presented in Sect. 1.2.1 and Sect. 1.2.2 is to protect the confidentiality of the message sent by Alice to Bob from a passive adversary. In the presence of an active adversary, who is able to modify the message in transit, these algorithms cannot guarantee that Alice indeed sent the message received by Bob. A message authentication code (MAC) algorithm is a symmetric-key algorithm used to assure the identity of the party from whom the message originated. If the original message sent by Alice got modified during transmission, she could not be considered as the originator of the altered message. Thus the MAC algorithm implicitly protects the data integrity too.

A MAC algorithm can be defined as a keyed hash function $h_K(\cdot)$, parametrised by a secret key K, which takes a message M of arbitrary length as input and outputs a fixed-length value often known as the MAC. It satisfies the following properties:

1: Calculation of $h_K(M)$ is computationally easy if K and M are known.

2: For an unknown K, given the MACs generated for any number of messages, it is computationally hard to find a valid MAC for any new message.

To protect the authenticity and integrity of the message M, Alice sends the MAC $t := h_K(M)$ along with the message, where K is only known to Alice and Bob. On receiving (M, t), Bob verifies the validity of the pair by recomputing the MAC. If Eve, who eavesdropped on the communication between Alice and Bob, can find another valid pair (M', t') without knowing K, such that $t' = h_K(M')$, then that pair is said to be a forgery. A good MAC algorithm should make it computationally infeasible to forge a valid MAC on a new message.

MAC algorithms are generally derived from block ciphers or CHFs, and due to this reason, relatively few dedicated MAC algorithms have been designed when com-

pared to other symmetric-key primitives. The Message Authenticator Algorithm (MAA) [50], which was part of the ISO 8731-2 banking standard [51], is one among them. In Sect.1.2.2, we saw the various modes of operations of block ciphers to perform encryption and authenticated encryption. Similarly, block ciphers can also be used in certain modes of operation to provide authentication. Some of the popular examples of block cipher based MACs are Cipher Block Chaining MAC (CBC-MAC) [52], encrypted CBC-MAC (EMAC) [53], eXtended Electronic Code Book MAC (XECB-MAC) [54], One-key MAC (OMAC) [55], Cipher-based MAC (CMAC) [56], Galois MAC (GMAC) [57] and Parallelizable MAC (PMAC) [58].

Though it is easy to compute the MAC using a keyless CHF by appending the secret key to the message, either as a prefix or suffix, such schemes are not secure against forgeries [59]. For instance, let h be an iterative CHF described in Sect. 1.1.3. For a message x, if the secret prefix method is used to compute the MAC $t := h(K \parallel x)$, it will be trivial to forge a MAC for some $x \parallel y$ using $h(t \parallel y)$ under the assumption that the CHF is vulnerable to length extension attack.[1] To preclude the forgery attacks, a combination of the secret prefix and secret suffix methods, known as the envelope method [59], was proposed. Later, to achieve even better security, a generic construction called MDx-MAC was proposed, which transforms any secure hash function of the MD4 family into a secure MAC algorithm by adding a key to the additive constants of the compression function [60]. The most popular CHF based MAC in use today is the HMAC, and we shall discuss it in more detail.

HMAC. The hash based MAC (HMAC) is a simple construction proposed by Bellare et al. in 1996 to generate the MAC using a CHF [61]. It has been standardised by NIST [62] and ISO [63]. Presently, it is used as a data authentication mechanism for the IPsec [64], SSH [65] and TLS [66] protocols. It employs an iterative hash function h, which uses the compression function f, in conjunction with a secret key K and generates a

[1] A length extension attack enables an attacker to compute $h(x \parallel y)$ for an unknown x and a chosen y using $h(x)$ and the length of x.

MAC for the message M as follows:

$$t = h((K_0 \oplus opad), h((K_0 \oplus ipad), M)),$$

where t is the MAC, $opad$ and $ipad$ are public constants, and K_0 is the secret key if $|K|$ equals the block size of h, or a function of K otherwise. The constants $opad$ and $ipad$ are generated by repeating the bytes $0x36$ and $0x5c$, each b times respectively, where b is the block size of the CHF in bytes. The different realisations of HMAC are denoted by HMAC-X, where X is the underlying CHF. Bellare et al. have proved that HMAC is a secure MAC algorithm if the following conditions hold [61]:

1: the CHF h is collision resistant for random and secret IVs

2: the compression function $f(\cdot)$ is pseudorandom[2]

3: the MAC computed using the compression function — $f(K, M)$ — is secure

According to the designers of HMAC, one of the advantages of HMAC is that if an implementation of the CHF is readily available, then "the MAC can be implemented by simply calling the existing function" [61]. Therefore, vulnerabilities in the CHF implementation may affect the HMAC constructed from it, and this will be demonstrated in Chapter 5.

1.2.4 Authenticated Encryption Algorithms

We have been discussing symmetric-key primitives that can encrypt or authenticate data. For a long time, it was believed that by using a strong encryption algorithm, we could also ensure the authenticity of data [35, 67]. Nevertheless, various studies have shown that one should never use encryption without providing authentication as an adversary can gather information about the secret key / plaintext or hijack a valid session by sending ciphertext of her choice to the decryption algorithm [68, 69]. For example, let us assume

[2]According to Bellare et al., "relatively weak form of pseudorandomness" suffices.

that Eve eavesdropped on the ciphertext $C_0, C_1, \ldots, C_{n-1}$ sent by Alice to Bob. Suppose she could establish a legitimate connection with Bob's cryptosystem, which uses a block cipher in the CBC mode. In that case, the last $n-1$ plaintext blocks sent by Alice can be successfully recovered by requesting the decryption for $C_0, C_1, \ldots, C_{n-1}$ [68].

There are three approaches to combining a secure encryption algorithm with a secure MAC algorithm to simultaneously provide confidentiality and authenticity:

- Encrypt-then-MAC (EtM), which generates MAC from the ciphertext,
- Encrypt-and-MAC (E&M), which generates ciphertext and MAC from the plaintext independently, and
- MAC-then-Encrypt (MtE), which generates MAC from the plaintext and then encrypts both of them.

These three approaches are illustrated in Figures 1.7, 1.8 and 1.9, respectively.

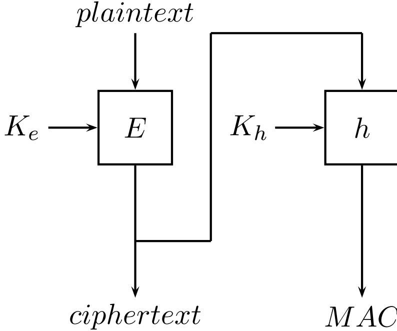


Figure 1.7: Encrypt-then-MAC (EtM) approach to combine encryption and MAC algorithms

Bellare and Namprempre analysed the three approaches and showed that EtM provides security against adaptive chosen ciphertext attack, provided that the underlying MAC algorithm is "strongly unforgeable" [70]. Though simplistic, these methods are inefficient as data has to be processed twice. Some of them are also prone to certain attacks due to incorrect usage [71]. To overcome these drawbacks, dedicated algorithms / primitives have been designed to perform authenticated encryption.

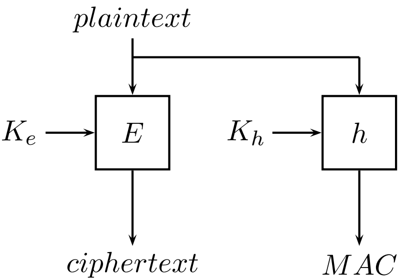


Figure 1.8: Encrypt-and-MAC (E&M) approach to combine encryption and MAC algorithms

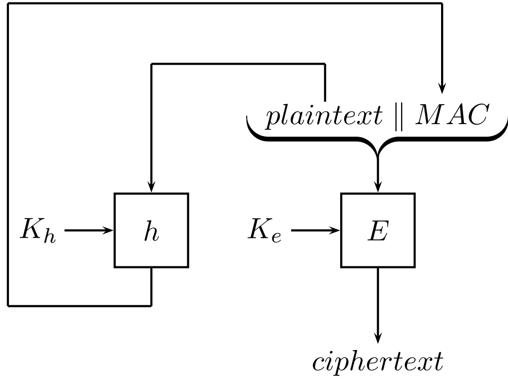


Figure 1.9: MAC-then-Encrypt (MtE) approach to combine encryption and MAC algorithms

Symmetric-key algorithms used to ensure a message's confidentiality and authenticity are called Authenticated Encryption (AE) algorithms. During encryption, an AE algorithm takes message M and key K as inputs and generates a ciphertext-MAC pair (C, T). During decryption, it takes (C, T) and K as inputs and outputs M after verifying the validity of (C, T); the ciphertext-MAC pair is discarded if found invalid. Sometimes, the message may contain some associated data or header A, which is not private but requires authentication. We use a variant of the AE known as Authenticated Encryption with Associated Data (AEAD) to process such messages. For an AEAD algorithm, C and T depends on (M, K) and (M, A, K), respectively. Thus it enables Bob to check the integrity of both the encrypted and unencrypted information in the data sent by Alice.

An authenticated encryption algorithm can be designed in three ways:

1. Use a block cipher in some special mode of operation. CCM [46], EAX [47], GCM [48], IAPM [72], XCBC [54] and OCB [45] modes are some of the AE / AEAD algorithms that use a block cipher as the underlying primitive.
2. Build it from a stream cipher so that the keystream is divided into two parts: one for encryption and another for authentication. A well-known example of such a design is the algorithm Grain-128a [73].
3. Design dedicated primitives that perform authenticated encryption. HELIX [26], ASCON [74], ACORN [75], AEGIS-128 [76] and Grain-128AEAD [77] are some popular examples of AE / AEAD primitives.

A detailed study of various authenticated encryption algorithms is not in the scope of this thesis.

1.3 Shannon's Theory of Secrecy

In 1948, with his landmark paper, 'A Mathematical Theory of Communication' [78], Shannon laid the foundations of information theory. In a follow-up paper, 'Communication Theory of Secrecy Systems' [14], he discussed cryptography from the viewpoint of information theory. Through these papers, he discussed the notion of perfect secrecy of a cryptosystem and the uncertainty in detecting its key. In this Section, we will briefly discuss some of the main ideas introduced by Shannon, which greatly influenced the scientific study of cryptography.

1.3.1 Perfect Secrecy

Let $\mathcal{P}$, $\mathcal{C}$ and $\mathcal{K}$ respectively represent the plaintext space, ciphertext space and key space of a symmetric-key cryptosystem whose encryption and decryption functions for some $K \in \mathcal{K}$ are denoted by $e_K(\cdot)$ and $d_K(\cdot)$, respectively. Let P and C represent

the random variables associated with the plaintext and ciphertext, respectively. For the cryptosystem to have perfect secrecy, the *a posteriori* probability that the plaintext is x, after observing that the ciphertext is y, must be equal to the *a priori* probability that P equals x, i.e. $\Pr(P = x|C = y) = \Pr(P = x) > 0$ for all $x \in \mathcal{P}$, $y \in \mathcal{C}$. An attacker gets no extra information about the plaintext by intercepting the ciphertext from such a cryptosystem.

Applying Bayes' theorem on the condition $\Pr(P = x|C = y) = \Pr(P = x) > 0$, Shannon stated that a necessary and sufficient condition for perfect secrecy is $\Pr(C = y|P = x) = \Pr(C = y) > 0$ for all $x \in \mathcal{P}$, $y \in \mathcal{C}$. The condition $\Pr(C = y|P = x) > 0$ also implies that for a fixed plaintext x, there must be at least one key that encrypts it to each ciphertext $y \in \mathcal{C}$; it follows that $|\mathcal{K}| \geq |\mathcal{C}|$. As the mapping from plaintext space to ciphertext space is injective, we get $|\mathcal{C}| \geq |\mathcal{P}|$.

Thus Shannon proved that to provide perfect secrecy the cryptosystem has to satisfy the condition $|\mathcal{K}| \geq |\mathcal{C}| \geq |\mathcal{P}|$. If $|\mathcal{P}| = |\mathcal{C}| = |\mathcal{K}|$, perfect secrecy will be possible if and only if every key is used with equal probability $1/|\mathcal{K}|$ and there is a unique K for every $x \in \mathcal{P}$, $y \in \mathcal{C}$ such that $y = e_K(x)$. As mentioned in Sect. 1.2.1, the Vernam cipher — popularly known as the one-time pad — is a well-known realisation of perfect secrecy.

1.3.2 Entropy and Redundancy

In Sect. 1.3.1, we saw that a cryptosystem could provide perfect secrecy only if its key space is at least as large as its plaintext space. It implies that a key must be used only for one encryption. But in real life, having a key space as large as the plaintext space will be difficult. To study the effect of encrypting multiple plaintexts using the same key, we must know the concepts of entropy and redundancy introduced by Shannon.

Let X be a discrete random variable which takes values from the set $\{x_0, x_1, \ldots, x_{n-1}\}$ such that $\Pr(X = x_i) = p_i$ for $i = 0, 1, \ldots, n-1$. The entropy of X, which is a mathematical measure of the information or uncertainty in X, is given

by $H(X) = -\sum_{i=0}^{n-1} p_i \log_2 p_i$. The conditional entropy of X given Y, where Y is another random variable, is given by $H(X|Y) = -\sum_y \sum_x \Pr(X = x|Y = y) \log_2 \Pr(X = x|Y = y)$.

Let M be the set of letters that constitutes a natural language L. The entropy of a random language formed from M is given by $\log_2 |M|$. If the entropy of L is H_L then the redundancy of the language L, which provides the fraction of letters in L that do not hold any information compared to a random language, is given by $R_L = 1 - \frac{H_L}{\log_2 |M|}$. For example, let $M = \{a, b\}$ such that $\Pr(X_L = a) = 0.25$ and $\Pr(X_L = b) = 0.75$, where X_L is a discrete random variable that belongs to the language L. Then, the entropy of L is given by $H_L = -(0.25 \log_2 0.25 + 0.75 \log_2 0.75) = 0.81$ and the redundancy of L is given by $R_L = 1 - \frac{0.81}{\log_2 2} = 0.19$.

1.3.3 Spurious Keys and Unicity Distance

Let K, P and C represent the random variables associated with the key, plaintext and ciphertext, respectively, of the aforementioned symmetric-key cryptosystem such that $K \in \mathcal{K}$, $P \in \mathcal{P}^n$ and $C \in \mathcal{C}^n$; in other words, the cryptosystem encrypts a plaintext string of length n using a single key. For the cryptosystem to have perfect secrecy, an attacker must not obtain any extra information about P by observing C, i.e. $H(P|C) = H(P)$. Shannon showed that the amount of uncertainty of the key remaining after knowing the ciphertext is given by $H(K|C) = H(K) + H(P) - H(C)$. The conditional entropy $H(K|C)$ is known as the *key equivocation*. For an ideal cipher, $H(P) = H(C) \implies H(K|C) = H(K)$.

The uncertainty in determining the key, even when the ciphertext is known, results in *spurious keys*. If the ciphertext space and plaintext space are equal in size, the expected number of spurious keys, s_n, given a ciphertext string of length n, where n is sufficiently large, satisfies the condition $s_n \geq \frac{|\mathcal{K}|}{|\mathcal{P}|^{nR_L}} - 1$, where R_L denotes the redundancy of the plaintext language. The expected number of spurious keys becomes 0 when n equals the *unicity distance* which is given by $n_0 \approx \frac{\log_2 |\mathcal{K}|}{R_L \log_2 |\mathcal{P}|}$. Therefore, a unique key can be

determined if we have a ciphertext string of length n_0.

The concept of spurious keys shows that key size alone does not guarantee the security of a block cipher, if an exhaustive key search is possible for the attacker with infinite computational power. Hence, the cipher has to be used within its unicity distance to protect from such adversaries.

1.4 Importance of the Secret Key

The secret key is the most critical parameter of any keyed cryptographic algorithm. The first known documentation of the importance of the key was by Kerckhoffs in 1883. According to him, in a cryptosystem, the only parameter that should be kept secret is the key; this is known as the *Kerckhoffs' principle*. The security of a cryptographic algorithm is quantified as the logarithmic measure of the fastest known attack against it. If an algorithm provides n-bit security, the complexity of the fastest known attack will be $O(2^n)$. Based on *Kerckhoffs' principle*, key size defines the upper bound — due to exhaustive key search — on the security of a well-designed symmetric-key primitive. For an ideal cipher, the lower bound on its security is also defined by the key size, since exhaustive key search is the fastest known attack against it.

Since symmetric-key algorithms rely on computational security, the time required to recover the secret key using the exhaustive key search or brute force attack will be exponentially proportional to the key size, under the assumption that the secret key was chosen uniformly at random from the entire key space. Taking into account computational resources available today, Moore's law [79] and advances in quantum computing, most of the symmetric-key algorithms use keys of size ranging from 128 bits to 256 bits. In contrast, PKC algorithms require longer keys to achieve equivalent security. When the key size is small, the mathematical problems they are based on are faster to solve than recovering the key using brute force. Therefore, to achieve 112-bit security, as recommended by NIST, an RSA key should be 2048 bits long, and an ECC key should be

224 bits long [80].

If the entire key or a part of it can be guessed, an attacker will be able to recover it with a complexity lesser than that of the exhaustive search. To preclude it, the keys generated should be distributed uniformly at random. For this reason, keys used for cryptographic applications are generated using cryptographically secure PRNGs — stream ciphers, block ciphers in counter mode, CHFs or HMACs — which are instantiated using the seeds generated from some random source having sufficient entropy [81, 82]. NIST recommends the use of PRNGs based on CHF, HMAC and AES in counter mode for key generation [81].

Whenever we discuss key generation in this thesis, the reader can assume that it uses a cryptographically secure PRNG.

1.5 Lightweight Cryptography

The twenty-first century is witnessing the dawn of a new class of computing environments where interconnected resource-constrained devices work in unison to carry out specific tasks. Examples of these environments include sensor networks, the Internet of Things (IoT), healthcare devices, distributed control systems, cyber-physical systems and automotive systems. Since conventional cryptographic algorithms are designed for desktop / server environments, using them in constrained devices may not always be possible due to the limited resources like CPU, memory and power. Owing to limited processing power and unavailability of complex instructions, 4-, 8- and 16-bit microcontrollers require a high number of CPU cycles to execute conventional cryptographic algorithms. Similarly, these devices may have minimal RAM and ROM, making them unsuited to run an algorithm with a large internal state or code size. Due to limited power availability, RFID tags, which are one of the most constrained devices, require cryptographic implementations with a very small amount of gate equivalents (GEs), and meeting stringent power and timing requirements. In order to cater to the information security

requirements of such highly constrained devices, lightweight cryptography (LWC) was introduced.

The performance of an LWC algorithm is expressed in terms of latency, throughput and power consumption. Likewise, its resource requirement is measured based on the RAM, ROM and CPU register requirements for software applications, and the gate area for hardware applications. Over the last decade, many LWC algorithms have been designed, and they aim to achieve a tradeoff between the performance and the resource requirement for a given level of security. Some of the symmetric-key ciphers discussed in this thesis are meant for lightweight applications. That being the case, it will be appropriate to briefly discuss the commonly followed design principles of lightweight stream ciphers and lightweight block ciphers.

1.5.1 Lightweight Stream Ciphers

Most of the lightweight stream ciphers in use today are based on a construction called the *Feedback Shift Register (FSR)*. An FSR of length l consists of l stages, which are updated during each cycle as follows:

1: A new word w is computed using the contents of a predefined set of stages which are often known as the *taps*.

2: The contents of each stage are shifted to their adjacent stages in such a way that the oldest element in the array will be output.

3: The first stage, which is empty after the shift operation, will be updated with the new element w.

An FSR that uses a linear function to compute w is known as the Linear Feedback Shift Register (LFSR). If the function is nonlinear, we get the Nonlinear Feedback Shift Register (NFSR). Another, but seldom used FSR is the Feedback with Carry Shift Register (FCSR).

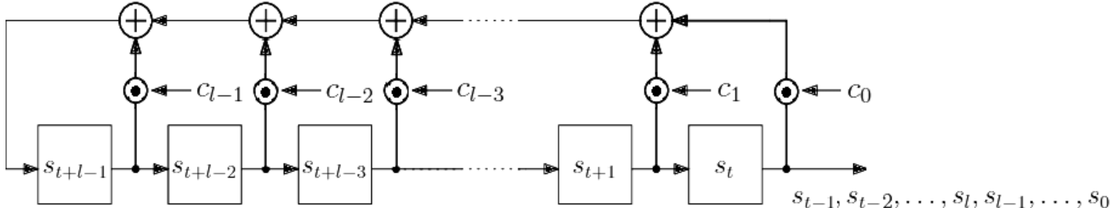


Figure 1.10: An LFSR of length l whose taps and stages (at time t) are represented by c_i and s_{t+i}, respectively, where $l-1 \leq i \leq 0$

LFSRs are more commonly used in constructing lightweight stream ciphers than the other FSR variants. The ciphers A5/1 [17], E0 [16], Grain [83] and MICKEY v2 [84] are some of the popular examples of LFSR-based lightweight constructions. The elements of an LFSR belong to the field $\mathbb{F}_q$, where the commonly used q equals 2^k for some integer $k \geq 1$. If $f(x)$, having coefficients in $\mathbb{F}_2$, is a primitive polynomial over $\mathbb{F}_{2^k}$ and α is its primitive root, then the elements of $\mathbb{F}_{2^k}$ can be denoted as $0, 1, \alpha, \alpha^2, \ldots, \alpha^{2^k-2}$. The feedback polynomial, which defines how an LFSR of length l is updated, can be represented as $x^l + \sum_{i=0}^{l-1} c_i \cdot x^i$, where $c_i \in \mathbb{F}_{2^k}$; $c_i \neq 0$ indicates the taps. The primitive polynomial, primitive root and feedback polynomial collectively define an LFSR. The schematic representation of an LFSR is given in Figure 1.10.

Though LFSRs are well-suited for hardware applications, they cannot be directly used as keystream generators because of their linear behaviour. Also, an adversary can quickly recover the internal state of an LFSR if she observes l output words. One classical method adopted to overcome this issue is to use a nonlinear combination function with high nonlinearity and correlation immunity to generate the keystream word from the outputs of multiple LFSRs. For example, the stream cipher E0 generates a keystream bit by combining the outputs of four LFSRs using a nonlinear FSM. The stream cipher Grain, which has a nonlinear function that mixes the outputs of an LFSR and an NFSR to generate the keystream bit, is also based on a similar design.

In yet another design, the nonlinear filter function generates the keystream word by mixing some of the bits of the LFSR; a well-known example is the cipher SNOW 3G [20]. Contrary to the methods mentioned above, there is another widely used strat-

egy that does not require nonlinear functions. In such designs, LFSRs are irregularly clocked to decimate their output sequences which will, in turn, introduce nonlinearity. The ciphers A5/1 and MICKEY v2 use irregularly clocked LFSRs.

The lightweight stream cipher ChaCha [85], a variant of the stream cipher Salsa20 [86], follows a different design principle that makes it optimised for software. It generates a block of the keystream by hashing the key, nonce, and respective block number using a hash function. The counter mode of operation, which helps to parallelise the keystream generation, and the use of simple operations such as modular addition, bitwise rotation and XOR enable the cipher to perform efficiently in software.

1.5.2 Lightweight Block Ciphers

Compared to ordinary block ciphers, the lightweight ones have some of the following features:

1: To reduce the memory usage, most of them support smaller block sizes and shorter keys.

2: The round functions will be using simple operations which need less ROM, RAM and CPU for software-oriented designs, and less gate area for hardware-oriented designs. To achieve the required security level, lightweight algorithms might need more number of rounds than the ordinary algorithms.

3: They will employ simple KSAs for the generation of subkeys to reduce the latency, memory and power consumption. In some designs, the round functions used in encryption will be reused to generate the subkeys.

4: They may have very similar encryption and decryption functions to reduce the code size. This can be achieved either by using involutions in the design [87], or by using encryption and decryption functions that are almost identical except for the key schedule [88].

Owing to the influence of AES [30], many lightweight block ciphers are based on

the SPN construction. Such ciphers rely on a component called the substitution-box (*S-box*), which performs substitution, to add nonlinearity to the round functions. Since implementing S-boxes as look-up tables require additional memory or hardware footprint, lightweight designs use S-boxes that are smaller than the ones used in ordinary algorithms. The cipher PRESENT, which is an ISO/IEC lightweight block cipher standard, is a notable example of the SPN-based design optimised for the hardware [89, 90]. It can encrypt plaintext blocks of size 64 bits using an 80- or 128-bit key, and uses a 4-bit S-box that can be efficiently implemented in hardware.

The other major category of lightweight block ciphers use FN-based constructions that we discussed in Sect. 1.2.2. The block cipher KASUMI, the confidentiality algorithm employed in 3G mobile communications, is an 8-round Feistel cipher whose block and key sizes are 64 and 128 bits, respectively [40]. A few other ciphers like CLEFIA [91] (an ISO/IEC lightweight block cipher standard [90]), and HIGHT [92] (an ISO/IEC block cipher standard [93]), use generalised FNs which have more than two branches. In lightweight designs, the round function of an FN is usually built using a small SPN as in Piccolo [94] and LBlock [95], or with simple arithmetic and logical operations, as in SIMON [96].

Certain lightweight ciphers, irrespective of having SPN- or FN-based constructions, are designed only using the following simple operations: modular addition, rotation and XOR. The designs based on ARX — standing for addition-rotation-XOR — depend on modular addition as the only source of nonlinearity. According to Dinu et al., “ARX and ARX-like designs are not only very fast, but also extremely small in terms of RAM footprint and code size” [97]. As per the FELICS framework that benchmarked the lightweight block ciphers on the microcontroller platforms 8-bit AVR, 16-bit MSP430, and 32-bit ARM, the ARX-based lightweight block ciphers Chaskey [98], SPECK [96] and LEA [99] have the most efficient software implementations on small processors [97]. Besides block ciphers, ARX-based designs have been adopted in the construction of stream ciphers such as Salsa20 [86] and ChaCha [85], and CHFs like the SHA-3

finalists Skein [100] and BLAKE [101].

1.6 Cryptanalysis

Cryptanalysis is the study of cryptosystems and their implementations to detect weaknesses that compromise information security functions. Cryptanalytic attacks on symmetric-key algorithms (including MAC algorithms) equip the adversary Eve to typically achieve one or more of the following goals.

1: Recovering the secret key.

2: Recovering the internal state.

3: Telling apart a cipher from an ideal or random source.

4: Obtaining plaintext information from the ciphertext.

5: Detecting structural weaknesses in algorithms.

6: Forging a valid MAC on an arbitrary message without the knowledge of the secret key.

The brute-force attack or exhaustive key search, the most naive way of recovering the key, defines the upper bound for the key recovery attacks on symmetric-key algorithms. If enough ciphertext / plaintext bits are unavailable, a brute-force attack can output incorrect keys. In the case of a stream cipher with a k-bit key, the probability that an incorrect key that can generate an n-bit keystream exists is given by $(2^k - 1) \cdot 2^{-n}$, under the assumption that the keystream bits are distributed uniformly at random. Therefore, an attacker with infinite computational power needs a t-bit keystream — obtained from t-bit plaintext-ciphertext pair — such that $2^{k-t} \approx 0$, to recover the key by exhaustive key search. Applying the same concept to a block cipher whose key size and block size are k and b bits, respectively, a brute-force attack is possible if l plaintext-ciphertext pairs are available such that $2^{k-lb} \approx 0$. The attack is also possible if the attacker knows

l_0 ciphertext blocks generated from some plaintext blocks containing redundancy, where l_0 is the unicity distance of the cryptosystem.

Stream ciphers and block ciphers might be also vulnerable to attacks that tell apart the ciphers from an ideal cipher; such attacks are known as distinguishing attacks. A classic example of a distinguishing attack is presented in [102]. Mantin and Shamir found that the second byte of the keystream generated by RC4 equals zero with twice the expected probability. The consequent distinguisher was strong enough to mount a practical attack on RC4 in some broadcast applications.

Cryptanalysis can be broadly classified into three types: algorithmic cryptanalysis, side-channel analysis and fault analysis. The former category, which has a weaker adversarial assumption than the others, exploits the weaknesses in the cryptographic algorithms. Side-channel analysis targets the weaknesses in cryptographic implementations and exploits the information that leaks during their normal functioning. In side-channel analysis, the adversary is often considered to be passive; i.e., she is merely able to collect side-channel information by monitoring the encryption / decryption device or the communication channel. Nevertheless, in the presence of an active adversary, who can manipulate the internal state or operations of the algorithm, cryptographic implementations may be vulnerable to fault attacks. In this thesis, attacks belonging to all these categories will be presented.

Depending on the ciphertext / plaintext data available to the adversary, cryptanalysis of symmetric-key ciphers follow one of the following models:

1: **Ciphertext-only attack.** As the name suggests, these attacks are built solely using the ciphertext and statistical information on the plaintext. As the entire ciphertext is available to the attacker, this is considered a strong attack and weak algorithms are vulnerable to such attacks.

2: **(Adaptive) chosen ciphertext attack.** In this attack model, Eve can decrypt ciphertexts of her choice. If her choice depends on the plaintext-ciphertext pairs already collected, it is called adaptive chosen ciphertext attack.

3: **Known plaintext attack.** This is the class of attacks in which the attacker has access to some plaintexts and the corresponding ciphertexts. Such attacks are possible only when some parts of the message are known to Eve.

4: **(Adaptive) chosen plaintext attack.** If Eve can encrypt plaintexts of her choice, she will be able to construct such attacks. In adaptive chosen plaintext attacks, new choices depend on the available plaintext-ciphertext pairs.

5: **(Adaptive) chosen plaintext and (adaptive) chosen ciphertext attack.** In this attack model, the adversary can encrypt and decrypt chosen plaintexts and ciphertexts, respectively.

A detailed discussion on various techniques used for cryptanalysis is not in the scope of this thesis. Nevertheless, we briefly discuss a few popular techniques used for the cryptanalysis of symmetric-key algorithms, which will help the reader comprehend the thesis well.

1.6.1 Algorithmic Cryptanalysis

Algebraic attack. Ciphers whose operations can be represented as a system of multivariate algebraic equations are found to be vulnerable to algebraic attacks. The adversary solves the relations involving key bits and plaintext-ciphertext bits to recover the secret key. The attack complexity depends on the difficulty in solving the system of equations which, in turn, is determined by the number of equations and variables involved and the amount of nonlinearity in the system. Therefore, to preclude algebraic attacks, designers strive to make ciphers a large system of highly nonlinear equations. The reader may refer to [103, 104, 105, 106] for more details on algebraic attacks.

Correlation attack. LFSRs are often coupled with nonlinear filter functions or combination functions in stream cipher constructions. Correlation attacks, proposed by Siegenthaler [107], are known plaintext attacks that exploit the input-output correlations

in such functions to recover the internal states of the LFSRs. The correlations lead to linear approximations of the nonlinear functions, and the attacker who finds linear relations between the internal state and the keystream bits with non-negligible probabilities can construct a correlation attack. Therefore, filter or combination functions should have high nonlinearity and correlation immunity to preclude correlation attacks.

Differential cryptanalysis. Differential cryptanalysis was introduced by Eli Biham and Adi Shamir in the late 1980s as a new attack technique against DES-like cryptosystems [108]. Let x and y be two variables such that $y = f(x)$, where f is a nonlinear round function. Differential cryptanalysis studies the statistical properties of the input-output difference pair (Δ_x, Δ_y), where $y \oplus \Delta_y = f(x \oplus \Delta_x)$, to examine if f behaves ideally with respect to its distribution. In the ideal case, output differences have to be distributed uniformly at random for a given input difference. The attacker's objective is to trace a path of highly probable input-output difference pairs across multiple rounds of the cipher, termed as differential or differential characteristic. In the case of a block cipher, the differential attack is a chosen plaintext attack — the attacker chooses plaintext pairs corresponding to a given differential characteristic — that exploit the non-uniform distribution of the differential to recover the key. Though stream ciphers are also vulnerable to differential cryptanalysis, the attacks are based on related-key or related-IV settings which are considered as weak attacks when compared to the non-related-key/IV attacks [109]. Higher-order differential, truncated differential, impossible differential, and boomerang attacks are the generalisations or extensions of differential cryptanalysis used against block ciphers [110, 111, 112, 113].

Linear cryptanalysis. Differential and linear cryptanalysis are the two most widely used techniques to analyse block ciphers. The discovery of linear cryptanalysis for block ciphers is attributed to Matsui. In 1992, Matsui and Yamagishi presented a known plaintext attack on the block cipher FEAL using a variant of this technique

[114]. Later in 1993, Matsui constructed a linear attack in its current form on DES [115]. In this attack, the adversary constructs probabilistic linear equations (called linear approximations) relating plaintext, ciphertext and key bits. Given a linear approximation that holds with high probability, the attacker finds a highly probable solution for the key bits for some known plaintext-ciphertext pairs. The application of linear cryptanalysis on stream ciphers was first presented by Golić in 1994 [116]. While attacking stream ciphers, the adversary aims to find some linear relations of the non-uniformly distributed keystream bits. Such biased linear approximations are used to distinguish the keystream sequence from a random sequence using statistical tests. The distinguishing attack presented in this thesis follows a similar attack methodology. Such distinguishing attacks can be used to reduce the uncertainty of unknown plaintexts or recover the keystream generator's internal state in certain cases.

Integral cryptanalysis. As the name indicates, this technique can be seen as a dual to differential cryptanalysis. Differential cryptanalysis analyses the propagation of plaintext differences through multiple rounds of a cipher. Whereas integral cryptanalysis, proposed by Knudsen, explores the propagation of plaintext sums through a cipher and is particularly applicable to block ciphers having bijective components [117]. The plaintext sum is computed by XORing all possible plaintexts with a given set of constant bits. For instance, the attacker might use 16 chosen plaintexts whose all but 4 bits are the same. If the cipher is vulnerable to integral cryptanalysis, the XOR sum of the corresponding ciphertexts can be predicted. The attacker exploits this information to recover the secret key.

Though there are many more techniques to perform algorithmic cryptanalysis, we are limiting to those used in this thesis.

1.6.2 Side-Channel Analysis

In 1996, Kocher published the seminal paper on side-channel attacks, where he showed that the implementations of Diffie-Hellman, RSA, DSS and some other cryptosystems can be attacked by observing their execution time [118]. Since then, several attacks on cryptographic implementations have been introduced. Even when a cryptosystem is secure against algorithmic attacks, its implementation, be it software or hardware, might be vulnerable to side-channel attacks. Such attacks analyse information leakage inherent to the cryptographic implementations through different channels such as power consumption, timing variations, electromagnetic and acoustic emanations, data remanence, hardware vulnerabilities, and processor flags. Side-channel analysis is of concern because it might lead to practical attacks on cryptosystems that are algorithmically secure.

Power Analysis. One of the most efficient and commonly used side-channel attacks is power analysis. It analyses the power consumption of a cryptographic device while performing encryption. The power consumed by the device varies depending on the instructions executed. Therefore, various parts of the encryption can be distinguished just by observing the power consumption measurements. *Simple Power Analysis* (*SPA*), which can reveal the sequence of operations in encryption, can be used to break the unprotected implementations of RSA and DES whose execution path is data-dependent [119]. Minor variations in the power consumption also occur due to the manipulation of data during encryption. As it is difficult to interpret such variations directly from a single power consumption measurement, Kocher et al. proposed a new technique known as the *Differential Power Analysis* (*DPA*) [119]. It uses statistical analysis of a large number of measurements recorded while encrypting multiple plaintexts to recover the secret key. DPA is generally considered to be more powerful and harder to prevent when compared to SPA.

Timing Attacks. Certain cipher implementations might not perform encryption in constant time. The variations in the timing for different inputs are mainly attributed to cache hits,[3] branching and conditional statements, processor instructions that run in variable time, and performance optimisations. As mentioned earlier, Kocher introduced the concept of timing attacks against such implementations [118]. He showed that fixed Diffie-Hellman exponents and factor RSA keys could be recovered using timing attacks.

Electromagnetic Attacks. A side-channel attack that exploits correlations between secret data and variations in the electromagnetic radiation emitted from an encryption device is known as an electromagnetic attack. Similar to power consumption, electromagnetic radiation levels vary according to the operations performed. Therefore, cipher implementations with data-dependent operations, such as the square-and-multiply implementation of RSA, are vulnerable to electromagnetic attacks [120].

Acoustic Cryptanalysis. Acoustic emanations, typically caused by the voltage regulation circuits of the CPU, are correlated with system activity. The concept of acoustic cryptanalysis was first presented by Shamir and Tromer, who reported that different RSA keys have different acoustic fingerprints [121]. In [122], Genkin et al. demonstrated that 4096-bit RSA decryption keys could be extracted from laptop computers, within an hour, by analysing the sound generated by the computer during the decryption of some chosen ciphertexts. Electromagnetic and acoustic cryptanalysis enable attacks on cryptographic devices even from a distance.

Attacks on Memory. Data remanence, the residual data retained in a digital memory after its intended lifetime, is another side-channel that leaks critical information. The data stored in Random Access Memory (RAM) chips are erased gradually on power loss. The duration of data retention in RAMs can be increased significantly by cooling

[3] A cache hit occurs when the data accessed by an instruction is found in the cache memory of the CPU, which stores copies of recently accessed data to serve them faster in future.

them to very low temperatures [123]. This behaviour of SRAMs and DRAMs leads to side-channel attacks, popularly known as cold boot attacks, which can be used to recover the keys or secret internal states of the ciphers [124]. An attacker can extract secret data from memory using malicious codes too. Hardware vulnerabilities of modern processors such as Meltdown, Spectre and SplitSpectre enable a malicious application to access the memory space of a different application and read its secrets [125, 126, 127]. RAMBleed is another more recent exploit which can be used by malicious codes to read some of the bits in any DDR3 and DDR4 DRAM memories without accessing them [128].

Processor Flag Attacks. In [129], Kelsey et al. first proposed that processor flags can be exploited to build side-channel attacks. Nearly every modern microprocessor has the status register, a collection of flag bits that store information about the state of the processors and information on operations performed by their ALUs [130]. The side-channel attacks against Streebog [131], RC5 [129], Schnorr-based identification and signature schemes [132], RSA and ECC based cryptosystems with exponent randomization [133], IDEA [134] and Twofish [134] exploit the carry flag which is a flag bit that indicates carry overflow in unsigned integer arithmetic. The side-channel attacks presented in this thesis also depend on the information leakage through the carry flag.

Countermeasures. Power analysis can be precluded by introducing noise into the power measurements, equalising power consumption using dummy operations and choosing operations that leak less information in their power consumption. In order to prevent timing attacks, cryptographic implementations must not have operations that are temporally correlated to secret parameters. In the software implementations of cryptographic algorithms, conditional branching statements must not be used and the cache state must be normalised just before and just after the cryptographic operation to ensure constant-

time execution. Masking secret inputs can be adopted to prevent attackers from knowing the inputs to specific functions like modular exponentiation. Similarly, processor flags can also be masked using dummy operations to prevent attackers from exploiting them. Providing tamper-proof physical shielding to cryptographic devices is also a good measure to make various side-channel attacks infeasible. The reader may refer to [135, 136, 137, 138] for more details on various countermeasures against side-channel attacks.

1.6.3 Fault Analysis

An attack against a cryptographic implementation by an active adversary who injects faults into the system is known as a fault attack. The adversary analyses the errors in the output due to the fault injections to recover the secret internal state of the cipher. Although it was known since the 1970s that electronic devices and microprocessors are vulnerable to fault injections under specific extreme environments [139], intentional injection of faults to break a cryptographic system was first discussed by Boneh et al. in 1997 [140]. The introduction of glitches on the clock signal and spikes in the input voltage level are two non-invasive techniques used to inject faults. An attacker can also rely on optical fault injection techniques using laser beams or photo flashes if she chooses to use semi-invasive techniques. The reader may refer to [141] for more details on the techniques used to inject faults. Depending on the technique used to inject faults, different fault models can be identified. These models tell about the number of bits affected by the fault injection, type of the fault, control on the fault location and timing, and duration of the fault. Fault analysis of a cipher often discusses the deterministic or statistical procedure to exploit the erroneous outputs due to the faults injected, assuming that fault injections follow a specific fault model. The fault-assisted side-channel analysis presented in this thesis also focuses on analysing the errors to recover the secret key.

1.7 Motivation, Objective and Scope

The security claims of the cryptographic algorithms may not always be supported with conclusive evidence or proof; furthermore, certain weaknesses in the design might have also been overlooked. Third-party security evaluations help detect unforeseen vulnerabilities in cryptographic algorithms / implementations and help ascertain the level of confidence in their security.

Lightweight symmetric-key ciphers are gaining popularity, and many new designs have been proposed recently. The (ultra-)lightweight WG family of stream ciphers [142] and the SPECK family of lightweight block ciphers [96] are two such designs that are considered to be potential candidates to secure lightweight applications like RFID devices and 4G/5G networks. The former has been granted a US patent [142] and the latter is an ISO standard for RFID devices [143]. The importance of these ciphers gave us enough motivation to scrutinise them.

It was found that SPECK, which uses modular addition in its round function, is vulnerable to carry flag attacks. Since HMAC-Streebog [144], which is a family of MAC algorithms defined in the Russian cryptographic standard, uses modular addition, we were motivated to analyse its security too. Despite the use of modular addition in SPECK and HMAC-Streebog, to the best of our knowledge, there are no prior published results analysing the resistance of unprotected implementations of these algorithms to carry flag attacks.

We started the research by evaluating some of the pitfalls in probability calculations related to cryptanalysis. The outcome of this study gave us more insights into how to perform cryptanalysis without misjudging security aspects of the concerned algorithms. Due to the reasons stated earlier, the following symmetric-key cryptographic algorithms or their implementations were chosen for cryptanalysis: the (ultra-)lightweight WG family of stream ciphers, the SPECK family of lightweight block ciphers and the HMAC-Streebog family of MAC algorithms. Finally, we proposed protected implementations of a family of stream ciphers to demonstrate some countermeasures to the software side-

channel attacks that we had studied.

This research therefore had the following objectives:

1: To emphasise the importance of probability assumptions and computations in cryptanalysis.

2: To perform algorithmic cryptanalysis of the (ultra-)lightweight WG family of stream ciphers.

3: To perform side-channel analysis of the unprotected software implementations of the SPECK family of lightweight block ciphers and the HMAC-Streebog family of MAC algorithms.

4: To suggest ways to protect software implementations of the RCR ciphers with detailed security and performance evaluations.

1.8 Outline of the Thesis

A brief introduction to the Chapters discussing the outcome of this research is provided in this Section.

Chapter 2. Probability calculations for cryptanalysis have to be done with utmost care to avoid misjudging security aspects of the concerned algorithms. In this Chapter, a case study to highlight the importance of probability assumptions and computations in cryptanalysis is presented.

The stream ciphers RCR-64 and RCR-32 designed by Sekar et al. [145] are the most recent additions to the Py family of stream ciphers, originally designed by Biham et al. [25]. The ciphers are among the fastest stream ciphers on software. To the best of our knowledge, the only reported attacks on the ciphers are due to Ding et al. [146], published in the *Journal of Universal Computer Science*. Our review of these alleged attacks on the RCR ciphers shows that they are based on non-existent keystream biases

stemming from flawed probability calculations [147]. The flawed computations by Ding et al. and the non-existence of their keystream biases are established in this Chapter.

In the next two Chapters, we discuss some cryptanalytic results on lightweight symmetric-key ciphers.

Chapter 3. In this Chapter, our attacks on a family of hardware-efficient lightweight synchronous stream ciphers are presented.

The Welch-Gong (WG) family of stream ciphers include two subfamilies, which we call WG-A and WG-B, of patented (ultra-)lightweight ciphers designed by Gong et al. [142]. The Waterloo Commercialization Office, Canada, has included the WG-A in an RFID anti-counterfeiting system [148] and has proposed the WG-B for securing 4G/5G networks [149]. The WG-A and WG-B ciphers support 80- and 128-bit keys, respectively. We detect input-output correlations in the nonlinear transformations used by these ciphers. Exploiting these, we show distinguishing attacks that require, to nearly ensure success, between $2^{22.20}$ and $2^{29.07}$ keystream samples for WG-A and not more than $2^{56.84}$ keystream samples for WG-B. We are not aware of any prior attacks on these ciphers.

Chapter 4. The attacks presented in Chapter 3 are algorithmic. Whereas, in Chapters 4 and 5, we discuss side-channel attacks.

The side-channel analysis of SPECK — a family of software-efficient lightweight block ciphers — is presented in this Chapter. SPECK is developed by Beaulieu et al. of the NSA for the Internet of Things (IoT) [96]. It is an ARX-based design with a Feistel-like structure which supports keys of size ranging from 64 bits to 256 bits. SPECK has been standardised by ISO/IEC for radio frequency identification (RFID) devices [143]. It has drawn the attention of many cryptanalysts, and several cryptanalysis results have been published.

In this Chapter, we present carry flag attacks on the full SPECK ciphers. Depending

on the key size and block size, the complexities of our attacks, to nearly ensure success in key recovery, vary from 2^{59} time and 2^{14} data to 2^{227} time and 2^{62} data.

Chapter 5. Any discussion on symmetric-key algorithms will be incomplete without MAC algorithms. In this Chapter, we present fault-assisted side-channel attacks on a family of HMAC algorithms.

Streebog is a family of hash functions defined in the Russian cryptographic standard GOST R 34.11–2012 [150]. HMAC–Streebog, which is defined in RFC 7836, is a Streebog based message authentication code [144]. It supports keys of size ranging from 256 bits to 512 bits. We present fault–assisted side-channel attacks on HMAC–Streebog–256 and HMAC–Streebog–512 that can recover the keys in real time with $2^{12.98}$ and $2^{14.97}$ average number of fault injections, respectively, to ensure 95% success. The attacker is assumed to be able to simultaneously flip at the most 181 chosen bits of the inner hash if it is a 256–bit variant, and 361 chosen bits of the hash otherwise. Compared to existing fault attacks on HMAC–Streebog, our attacks have a larger temporal window for fault injection, target a more accessible location and cannot be mitigated with output redundancy countermeasures. Some of the latest hardware vulnerabilities make the HMAC–Streebog implementations vulnerable to our attacks.

Chapter 6. Following the discussions on cryptanalysis, we move on to the secure implementation of symmetric-key ciphers. We evaluate the security of the stream ciphers RCR–64 and RCR–32, and propose their protected implementations.

The synchronous stream ciphers RCR-64 and RCR-32 designed by Sekar, Paul and Preneel [145] are strengthened variants of the ciphers TPy and TPypy (designed by Biham and Seberry) [151], respectively. The RCR ciphers have remained unbroken since they were published in 2007. In this Chapter, we present arguments that not only support the designers' security claims but suggest, in general, that the ciphers are secure against several classes of cryptanalytic attacks. We find that the ciphers are

best used with 256-bit keys and 384-bit IVs. We also suggest ways to protect software implementations of the RCR ciphers against (cache-)timing and processor flag attacks. Our performance evaluation suggests that the protected implementation of the RCR-64 encrypts long messages at speeds comparable to some of the fastest stream ciphers available today. Consequently, we find that the RCR ciphers may be well suited for PC-based applications in general and streaming audio / video applications in particular.

Chapter 7. In this Chapter, we give our concluding remarks and some interesting problems for future work.

Chapter 2

On the Security of the Stream Ciphers RCR-64 and RCR-32

2.1 Introduction

The security bounds for cryptographic algorithms are generally determined after investigating their resistance to several cryptanalytic techniques such as linear cryptanalysis, differential cryptanalysis, integral cryptanalysis, correlation attacks, and so on. In the process, we frequently invoke the Bayes' theorem, assume that certain events are independent / mutually exclusive, etc. It goes without saying that such assumptions have to be justifiable. Disregarding the occurrence of certain events may also yield erroneous results. Hence, probability calculations for cryptanalysis have to be done with utmost care so that the security of the concerned algorithms are not misjudged.

The Py family of stream ciphers. The stream ciphers Py [25], Pypy [152], and Py6 [25], designed by Biham et al., are among the fastest eSTREAM Profile 1 candidates [153, 154]. The designers later changed the key schedule algorithms of these ciphers to produce strengthened variants, TPy, TPypy and TPy6, respectively [151]. As the new

variants were still vulnerable to cryptanalysis,[1] Sekar et al. proposed two new variants, RCR-64 and RCR-32 [145]. All these ciphers are recommended for use with 256-bit keys and 128-bit IVs. However, the size of the key may vary from 1 to 256 bytes (in steps of a byte) and the size of the IV from 1 to 64 bytes (in steps of a byte). The Py family of stream ciphers have been extensively analysed over time and RCR-64 and RCR-32, the most recent additions to the family, are conjectured to be the strongest of the lot.

Performance evaluation. The eBASC (ECRYPT Benchmarking of Stream Ciphers) project [155] has evaluated the performances of a few stream ciphers, including the TPy, TPypy and TPy6, on Intel Core i5-1030NG7 processor to encrypt 1536-byte, 4096-byte and much longer messages with 256-bit keys and the results are given in Table 2.1. We find the performances of TPy, TPypy and TPy6 compararable to those of Sosemanuk [156], an eSTREAM final portfolio cipher [157], and SNOW 2.0 [18], an ISO/IEC keystream generator standard [19], and better than that of the AES [30] in counter mode implemented without AES-NI instruction set [158]. Although the performances of the RCR-64 and RCR-32 have not been evaluated by eBASC project, it can be safely assumed that they will be marginally faster than the TPy and TPypy, respectively, due to the absence of three operations—an array access, an addition and a bitwise AND—per encryption round.

Flawed probability calculation. The *Journal of Universal Computer Science* published a paper presenting related-key distinguishing attacks on the stream ciphers Py, Pypy, TPy, TPypy, RCR-64 and RCR-32 [146]. To the best of our knowledge, the paper contains the only known attacks on the RCR-64 and RCR-32. Our review of the attacks

[1] The designers do not explicitly state in [151] that their security claims for Py, Pypy and Py6 apply to TPy, TPypy and TPy6, respectively. Even if the claims (that limit the number of keystream bytes to 2^{64}) are applicable, they do not specify the number of key/IV pairs that can be considered to construct meaningful attacks. Even supposing that the 2^{64} bound holds regardless of the number of key/IV pairs used, the ciphers TPy and TPypy may still be seen as vulnerable to cryptanalysis (in an academic sense) but not broken under the designers' claims.

reveal flaws in probability calculations, leading to the visualisation of biases where they do not exist. This Chapter presents our findings in detail. A note is also made of potential flaws in probability-based cryptanalysis.

Table 2.1: The performances of a few stream ciphers, including the AES in counter mode, to encrypt 1536-byte, 4096-byte and much longer messages with 256-bit keys on Intel Core i5-1030NG7 processor as measured by the eBASC project

Cipher	**Performance in cycles per byte to encrypt**		
	1536-byte message	**4096-byte message**	**long message**
ChaCha8 [85]	0.38	0.29	0.29
Salsa20/8 [86]	0.50	0.37	0.35
ChaCha12 [85]	0.51	0.40	0.40
Salsa20/12 [86]	0.64	0.48	0.46
AES[a] [30]	0.62	0.54	0.49
ChaCha20 [85]	0.79	0.60	0.60
Salsa20/20 [86]	0.95	0.69	0.67
HC-256 [23]	19.31	8.49	1.98
Sosemanuk [156]	2.92	2.40	2.15
TPy [151]	6.26	3.82	2.24
SNOW 2.0 [18]	2.56	2.39	2.30
TPy6 [151]	4.40	3.19	2.42
CryptMT v3 [159]	4.29	2.98	2.45
TPypy [151]	7.59	4.93	3.41
AES[b] [30]	13.98	13.87	13.80

[a] AES in counter mode implemented using AES-NI instruction set [160].
[b] AES in counter mode implemented without AES-NI instruction set [161].

Organisation of the Chapter. The remaining Chapter is organised as follows. In Sect. 2.2, we discuss cases where errors in probability assumptions or computations have led or can potentially lead to flawed cryptanalysis. The analysis of the alleged attacks on the RCR ciphers by Ding et al. is presented in Sect. 2.3. We conclude in Sect. 2.4.

2.2 Probability Assumptions and Computations in Cryptanalysis

2.2.1 Independent Events

If the probabilities of two events A and B are known, their joint probability can be computed as Pr$(A \cap B)$ $=$ Pr$(A) \cdot$ Pr(B) when A and B are independent. If they are dependent, then Pr$(A \cap B)$ $=$ Pr$(A \mid B) \cdot$ Pr(B) or Pr$(A \cap B)$ $=$ Pr$(A) \cdot$ Pr$(B \mid A)$. The assumption of independence is widely used in cryptanalysis to cascade the effects of multiple events which are generally related to different rounds or state variables. In many cases, it might be challenging to compute Pr$(A \mid B)$, and one may assume that A and B are independent. For instance, in the differential cryptanalysis [108] and linear cryptanalysis [115] on DES the differential and linear characteristics of the different rounds of DES are cascaded assuming independence of certain events [162], which is reasonable when the round keys are distributed uniformly at random. Harpes et al. used the example of two 2-bit adders cascaded together to show that Matsui's Piling-up Lemma [115] computes probability erroneously when a certain assumption of independence is invalid [163]. Therefore, without convincing evidence, two events must not be assumed to be independent.

2.2.2 Pairwise Disjoint Events

In cryptanalysis, we may come across certain occasions where an event under consideration (call it A) depends on several other events. If these events are pairwise mutually exclusive, by simply adding their probabilities, we obtain Pr(A). For instance, the cryptanalysis of the full Spritz stream cipher depends on the union of three pairwise disjoint events that guarantee that the first two output bytes produced by the cipher are equal to a certain value [164]. There may be scenarios where one has to assume mutual exclusivity of events in order to compute a certain probability. Such an assumption has to be

justifiable.

2.2.3 High Probability Events

In order to compute the probability of an event, it is very much necessary to identify all the events influencing it. In cryptanalysis, it may not always be possible to identify all such events due to the sheer complexity of the operations involved in mixing the elements of the internal state. For example, in the distinguishing attacks of [165] on the stream cipher HC-256, some events were ignored as they were experimentally found to occur with negligible probability when compared to events that were ultimately taken into consideration. Nevertheless, the knowledge of high probability events allow us to make reasonable approximations. For instance, let S_1, S_2 and S_3 be three pairwise disjoint events such that the occurrence of any of them results in the occurrence of another event A. If Pr(S_1) is much greater than Pr(S_2) or Pr(S_3), then it will evidently be erroneous to compute Pr(A) without considering S_1.

2.2.4 Uniformly Distributed Random Variables

Under the assumptions that the key is distributed uniformly at random, and that the key scheduling algorithm mixes the key well into the internal state of the cipher, it is generally assumed that the internal state, except for the constituent permutations, during the encryption phase is also distributed uniformly at random. Based on this assumption, the occurrence of biased outputs is examined. For example, let a, b and c be Boolean variables such that $c = a \oplus b$, and Pr$(b = 0) \neq 0.5$. The assumption that Pr$(a = 0) = 0.5$ will mean that c is an unbiased random variable. If Pr$(a = 0) \neq 0.5$, then it follows that Pr$(c = 0) \neq 0.5$. Even while analysing complex nonlinear functions, which may be the case more often, one must be very careful to assume that its output is uniformly distributed at random.

2.3 The Alleged related-key Distinguishing Attacks by Ding et al.

2.3.1 Specifications of the Py family of Stream Ciphers

Each member of the Py family of ciphers, including the variants RCR-64 and RCR-32, is composed of three parts: a key setup algorithm, an IV setup algorithm and a round function. The first two parts mix the secret key and the IV to generate an internal state composed of a permutation P of 256 elements, 260-element array Y where each element is a 32-bit word and a 32-bit variable s. The key/IV setup uses two intermediate variables: a fixed permutation of 256 elements denoted by IP and a variable EIV whose size is equal to that of the IV. The round function, which is executed iteratively, updates the internal state and generates pseudorandom keystream bits. In order to roll an n-element array S, the IV setup and the round function use the function rotate($\cdot$) that takes the array $\{S[0], S[1], \cdots, S[n-2], S[n-1]\}$ as input and outputs $\{S[1], S[2], \cdots, S[n-1], S[0]\}$. The key setup algorithms of the Py, the Pypy, the TPy, the TPypy, the RCR-64 and the RCR-32 are identical (see Algorithm 1). As to where the ciphers differ can be inferred from Table 2.2, Algorithm 2 and Algorithm 3. A visual representation of the round function is shown in Figure 2.1. The details of the algorithms can also be found in [25, 152, 151].

2.3.2 Description of Ding et al.'s Alleged Related-Key Attacks

Let k_1 and k_2 be two keys of size 256 bytes each such that, $k_1[16] \oplus k_2[16] = 1$, $k_1[17] \neq k_2[17]$ and $k_1[i] = k_2[i]$, for $i \neq 16, 17$ and $i < 256$. Let us assume that k_1 and k_2 are used with identical IVs to initialise the Py family of ciphers. When k_i is used, (Y_i^j, P_i^j, s_i^j) will be the internal state at the beginning of jth round of encryption, which generates Z_i^j as the second output word, for $i = 1, 2$. Let D denote the event $Y_1^1[i] = Y_2^1[i]$, for $-3 \leq i \leq 12$. The probability of occurrence of D has been computed by Sekar et al. as

$2^{-28.4}$ [145]. Ding et al. claimed distinguishers for Py, Pypy, TPy, TPypy, RCR-64 and RCR-32 based on the simultaneous occurrence of D and a few other events that they identified.

Table 2.2: Notation for different parts of the ciphers Py, Pypy, TPy, TPypy, RCR-64 and RCR-32

	Py	Pypy	TPy	TPypy	RCR-64	RCR-32
Key setup	$\mathcal{K}$	$\mathcal{K}$	$\mathcal{K}$	$\mathcal{K}$	$\mathcal{K}$	$\mathcal{K}$
IV setup	$\mathcal{I}_1$	$\mathcal{I}_1$	$\mathcal{I}_2$	$\mathcal{I}_2$	$\mathcal{I}_2$	$\mathcal{I}_2$
Round function	$\mathcal{R}_1$	$\mathcal{R}_2$	$\mathcal{R}_1$	$\mathcal{R}_2$	$\mathcal{R}_3$	$\mathcal{R}_4$

Algorithm 1 Key setup: $\mathcal{K}$

Require: A key, an IV and an initial permutation
Ensure: An array $Y[-3, -2, ..., 256]$

```
kb = size of the key k in bytes;
ivb = size of the IV in bytes;
L = -3;
H = 256;

s = IP[kb - 1];
s = (s << 8) | IP[(s  ^ (ivb - 1)) & 0xFF];
s = (s << 8) | IP[(s  ^ k[0]) & 0xFF];
s = (s << 8) | IP[(s  ^ k[kb - 1]) & 0xFF];

for(j = 0; j < kb; j++)
{
        s = s + k[j];
        s0 = IP[s & 0xFF];
        s = ROTL32(s, 8)  ^ (u32) s0;
}

for(j = 0; j < kb; j++)
{
        s = s + k[j];
        s0 = IP[s & 0xFF];
        s ^= ROTL32(s, 8) + (u32) s0;
}

/*Initialise the array Y*/
for(i = L, j = 0; i <= H; i++)
{
        s = s + k[j];
        s0 = IP[s & 0xFF];
        Y[i] = s = ROTL32(s, 8)  ^ (u32) s0;
        j = j + 1 (mod kb);
}
```

Algorithm 2 IV setup algorithms: $\mathcal{I}_1$ and $\mathcal{I}_2$

Require: The Y and the IV

Ensure: The arrays $Y[-3, -2, ..., 256]$, $P[0, 1, ..., 255]$ and the variable s

```
/*Create an initial permutation*/
ivb = size of IV in bytes; L = -3; H = 256;

v = IV[0] ^ ((Y[0] >> 16) & 0xFF);
d = (IV[1 (mod ivb)]  ^ ((Y[1] >> 16) & 0xFF)) | 1;

for(i = 0; i < 256; i++)
{
        P[i] = IP[v]; v += d;
}

/*Initialise s*/
s = ((u32) v << 24)  ^ ((u32) d << 16) ^ ((u32) P[254] << 8)
    ^ ((u32) P[255]);
s ^= Y[L] + Y[H];

for(i = 0; i < ivb; i++)
{
        s = s + IV[i] + Y[L + i];
        s0 = P[s & 0xFF]; EIV[i] = s0;
        s = ROTL32(s, 8) ^ (u32) s0;
}

/*Update EIV*/
for(i = 0; i < ivb; i++)
{
        /*Skip the next step for I1*/
        s += EIV[i + ivb - 1 (mod ivb)] + Y[H - i];
        /*Skip the next step for I2*/
        s += IV[i] + Y[H - i];
        s0 = P[s & 0xFF]; EIV[i] += s0;
        s = ROTL32(s, 8) ^ (u32) s0;
}

/*Update the rolling arrays and the variable s*/
for(i = 0; i < 260; i++)
{
        x0 = EIV[0] = EIV[0] ^ (s & 0xFF);
        rotate(EIV);
        swap(P[0], P[x0]);
        rotate(P);
        /*Skip the next two steps for I1*/
        s = ROTL32(s, 8) + Y[H];
        Y[L] += s ^ Y[x0];
        /*Skip the next step for I2*/
        Y[L] = s = (s ^ Y[L]) + Y[x0];
        rotate(Y);
}

s = s + Y[26] + Y[153] + Y[208];
if(s == 0) {s = (kb * 8) + ((ivb * 8) << 16) + 0x87654321;}
```

Algorithm 3 Round functions: $\mathcal{R}_1$, $\mathcal{R}_2$, $\mathcal{R}_3$ and $\mathcal{R}_4$

Require: $Y[-3,-2,...,256]$, $P[0,1,...,255]$ and a 32-bit variable s
Ensure: A pseudorandom output Z

```
swap(P[0], P[Y[185] & 255]);
rotate(P);

s = s + Y[P[72]] - Y[P[239]];
/*Skip the next step for R3 and R4*/
s = ROTL32(s, ((P[116] + 18) & 31));
/*Skip the next step for R1 and R2*/
s = ROTL32(s, 19);

/*Skip the next step for R2 and R4*/
Z ⟵ ((ROTL32(s, 25) ^ Y[256]) + Y[P[26]]);

Z ⟵ Z || ((s ^ Y[-1]) + Y[P[208]]);

Y[-3]=(ROTL32(s, 14) ^ Y[-3]) + Y[P[153]];
rotate(Y);
```

Alleged Related-Key Distinguishing Attacks on RCR-64 and RCR-32

The formulae for the least significant bit (LSB) of the output words Z_1^1, Z_1^2, Z_2^1, and Z_2^2 are as follows:

$$Z^1_{1(0)} = s^2_{1(0)} \oplus Y^1_1[-1]_{(0)} \oplus Y^1_1[P^2_1[208]]_{(0)}\,, \tag{2.1}$$

$$Z^2_{1(0)} = s^3_{1(0)} \oplus Y^2_1[-1]_{(0)} \oplus Y^2_1[P^3_1[208]]_{(0)}\,, \tag{2.2}$$

$$Z^1_{2(0)} = s^2_{2(0)} \oplus Y^1_2[-1]_{(0)} \oplus Y^1_2[P^2_2[208]]_{(0)}\,, \tag{2.3}$$

$$Z^2_{2(0)} = s^3_{2(0)} \oplus Y^2_2[-1]_{(0)} \oplus Y^2_2[P^3_2[208]]_{(0)}\,. \tag{2.4}$$

When event D occurs,

$$Y^1_1[-1]_{(0)} = Y^1_2[-1]_{(0)}\,, \tag{2.5}$$

$$Y^2_1[-1]_{(0)} = Y^2_2[-1]_{(0)}\,. \tag{2.6}$$

Let the event $Y^1_1[P^2_1[208]]_{(0)} \oplus Y^2_1[P^3_1[208]]_{(0)} \oplus Y^1_2[P^2_2[208]]_{(0)} \oplus Y^2_2[P^3_2[208]]_{(0)} = 0$ be denoted by G. If G_1 and G_2 denote the events $P^2_1[208] = P^3_1[208]+1$ and $P^2_2[208] =$

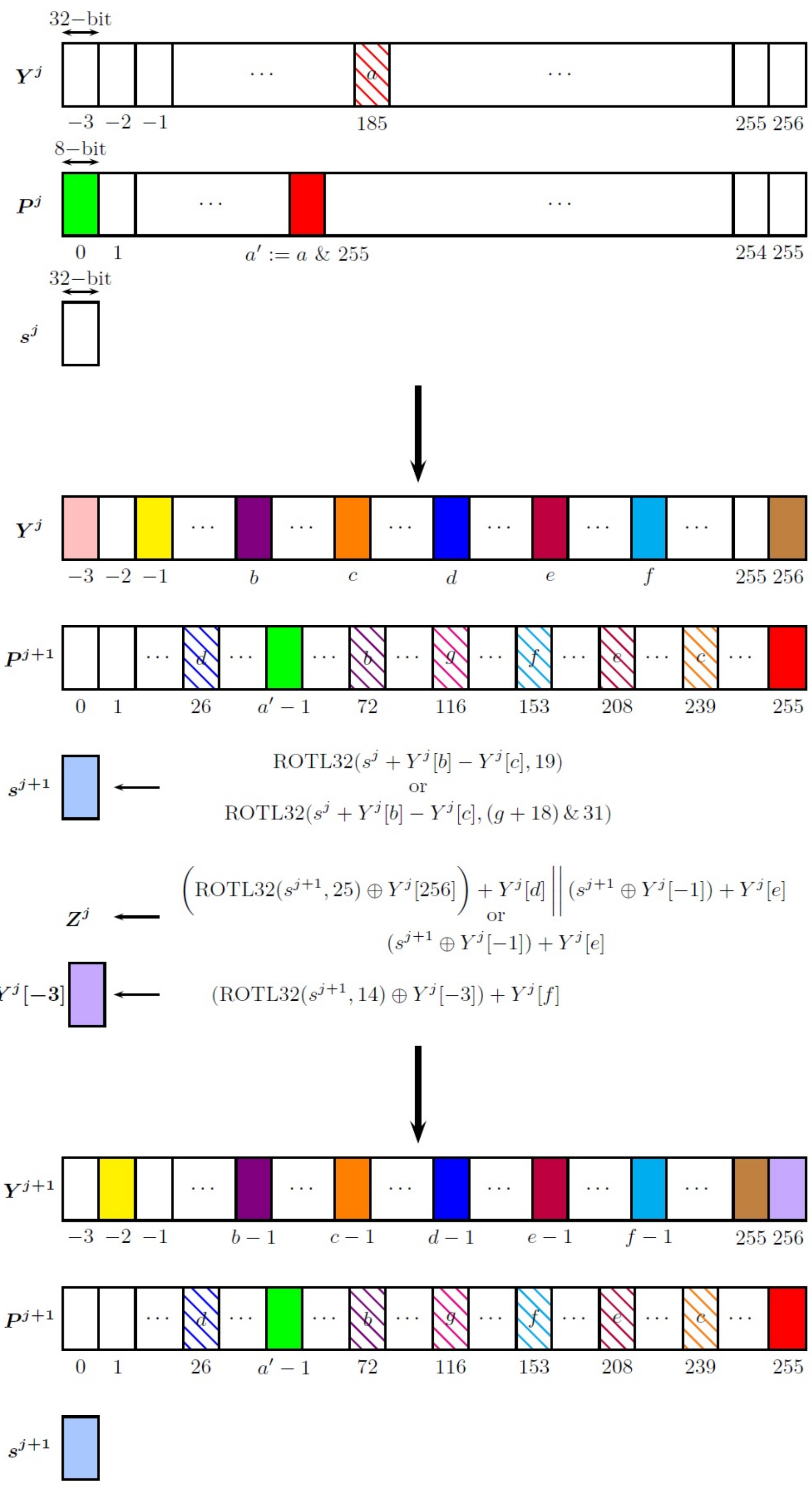


Figure 2.1: Visual representation of the round function of the Py family of ciphers, where (Y^j, P^j, s^j) and Z^j are the internal state at the beginning of jth round and the output word generated from it, respectively

$P_2^3[208]+1$, respectively, we have:

$$\begin{aligned} \Pr(G) &\approx \Pr(G_1 \cap G_2) \\ &\approx 2^{-16}. \end{aligned} \tag{2.7}$$

From Algorithm 3, we know that:

$$s_1^3 = ROTL32(s_1^2 + Y_1^2[P_1^3[72]] - Y_1^2[P_1^3[239]], 19)\,, \tag{2.8}$$

$$s_2^3 = ROTL32(s_2^2 + Y_2^2[P_2^3[72]] - Y_2^2[P_2^3[239]], 19)\,. \tag{2.9}$$

If $c_1 = Y_1^2[P_1^3[72]] - Y_1^2[P_1^3[239]]$, $c_2 = Y_2^2[P_2^3[72]] - Y_2^2[P_2^3[239]]$, and $\delta_{(13)}$ and $\gamma_{(13)}$ represent the carry bits generated at the 13th bit-position (where LSB is the 1st bit) in (2.8) and (2.9),[2] respectively, we get:

$$\begin{aligned} s_{1(0)}^2 \oplus s_{1(0)}^3 \oplus s_{2(0)}^2 \oplus s_{2(0)}^3 &= s_{1(0)}^2 \oplus s_{1(13)}^2 \oplus s_{2(0)}^2 \oplus s_{2(13)}^2 \oplus c_{1(13)} \oplus c_{2(13)} \oplus \\ &\quad \delta_{(13)} \oplus \gamma_{(13)}\,. \end{aligned}$$

Let us define the events E, E_1 and E_2 as follows:

- E represents the event $s_{1(0)}^2 \oplus s_{1(0)}^3 \oplus s_{2(0)}^2 \oplus s_{2(0)}^3 = 0$,
- E_1 represents the event $s_{1(0)}^2 \oplus s_{1(13)}^2 \oplus s_{2(0)}^2 \oplus s_{2(13)}^2 \oplus \delta_{(13)} \oplus \gamma_{(13)} = 0$,
- E_2 represents the event $P_1^3[72] = P_2^3[72] = a$ and $P_1^3[239] = P_2^3[239] = b$, where $-3 \le a, b \le 11$ and $a \ne b$.

Under the assumption that s_1^2 and s_2^2 are distributed uniformly at random and, P_1^3 and P_2^3 being permutations, Ding et al. computed the following:

$$\begin{aligned} \Pr(E_1) &= \frac{1}{2}\,, \\ \Pr(E_2) &= 2^{-23.7}\,. \end{aligned}$$

[2] In the original paper, Ding et al. considered 19th bit as LSB after the rotation, instead of 13th bit.

From Algorithm 3, we know that:

$$
\begin{aligned}
& Y_1^1[i] = Y_2^1[i] \implies Y_1^2[j] = Y_2^2[j]\,, \\
& \text{where} - 3 \leq i \leq 12 \text{ and} - 3 \leq j \leq 11\,.
\end{aligned} \tag{2.10}
$$

Hence, when event D occurs, we get:

$$
\Pr(E) = \Pr(E_1 \cap E_2) \approx \Pr(E_1) \cdot \Pr(E_2) = 2^{-24.7}\,. \tag{2.11}
$$

Ding et al. assumed that $Z^1_{1(0)} \oplus Z^2_{1(0)} \oplus Z^1_{2(0)} \oplus Z^2_{2(0)}$ is distributed uniformly at random when $D \cap G \cap E$ does not occur. Hence, under the assumption that D, G and E are independent events, it is obtained that:

$$
\begin{aligned}
\Pr(Z^1_{1(0)} & \oplus Z^2_{1(0)} \oplus Z^1_{2(0)} \oplus Z^2_{2(0)} = 0) \\
& = \Pr(D \cap G \cap E) + \frac{1}{2}\Big(1 - \Pr(D \cap G \cap E)\Big) \\
& = \Pr(D) \cdot \Pr(G) \cdot \Pr(E) + \frac{1}{2}\Big(1 - \Pr(D) \cdot \Pr(G) \cdot \Pr(E)\Big) \\
& = \frac{1}{2}(1 + 2^{-69.2})\,.
\end{aligned} \tag{2.12}
$$

The bias detected in $Z^1_{1(0)} \oplus Z^2_{1(0)} \oplus Z^1_{2(0)} \oplus Z^2_{2(0)} = 0$ as per (2.12) yielded the alleged related-key distinguishing attacks of [146] on RCR-64 and RCR-32.

Alleged Related-Key Distinguishing Attacks on Py, Pypy, TPy and TPypy

Event G, and equations (2.1)–(2.7), which were described in Sect. 2.3.2 for the ciphers RCR-64 and RCR-32, are valid for the ciphers Py, Pypy, TPy and TPypy as well. The ciphers Py, Pypy, TPy and TPypy differ from RCR-64 and RCR-32 in the update of s

and, from Algorithm 3, we get:

$$\begin{aligned} s_1^3 &= ROTL32\Big(s_1^2 + Y_1^2[P_1^3[72]] - Y_1^2[P_1^3[239]], \\ &\quad P_1^3[116] + 18 \,(\mathrm{mod}\ 32)\Big)\,, \qquad (2.13) \\ s_2^3 &= ROTL32\Big(s_2^2 + Y_2^2[P_2^3[72]] - Y_2^2[P_2^3[239]], \\ &\quad P_2^3[116] + 18 \,(\mathrm{mod}\ 32)\Big)\,. \qquad (2.14) \end{aligned}$$

Let $e_1 = Y_1^2[P_1^3[72]] - Y_1^2[P_1^3[239]]$, $e_2 = Y_2^2[P_2^3[72]] - Y_2^2[P_2^3[239]]$, $d_1 = 32 - P_1^3[116] + 18 \,(\mathrm{mod}\ 32)$, $d_2 = 32 - P_2^3[116] + 18 \,(\mathrm{mod}\ 32)$, and $\beta_{(d_1)}$ and $\varepsilon_{(d_2)}$ represent the carry bits generated at the d_1th bit-position in (2.13) and d_2th bit-position in (2.14), respectively.[3] We get:

$$s_{1(0)}^2 \oplus s_{1(0)}^3 \oplus s_{2(0)}^2 \oplus s_{2(0)}^3 = s_{1(0)}^2 \oplus s_{1(d_1)}^2 \oplus s_{2(0)}^2 \oplus s_{2(d_2)}^2 \oplus e_{1(d_1)} \oplus e_{2(d_2)} \oplus \beta_{(d_1)} \oplus \epsilon_{(d_2)}\,.$$

Let us consider the following events:

- F represents the event $s_{1(0)}^2 \oplus s_{1(0)}^3 \oplus s_{2(0)}^2 \oplus s_{2(0)}^3 = 0$,
- F_1 represents the event $d_1 = d_2$,
- F_2 represents the event $s_{1(0)}^2 \oplus s_{1(d_1)}^2 \oplus s_{2(0)}^2 \oplus s_{2(d_2)}^2 \oplus \beta_{(d_1)} \oplus \epsilon_{(d_2)} = 0$,
- F_3 represents the event $P_1^3[72] = P_2^3[72] = a$ and $P_1^3[239] = P_2^3[239] = b$, where $-3 \leq a, b \leq 11$ and $a \neq b$.

According to Ding et al.,

$$\Pr(F_1) = 2^{-5}\,,$$

[3] In the original paper, Ding et al. considered ith bit as LSB after the rotation, instead of $(32-i)$th bit, where $i = P_j^3[116] + 18 \,(\mathrm{mod}\ 32)$ for $j = 1, 2$.

$$\Pr(F_2) = \frac{1}{2},$$
$$\Pr(F_3) = 2^{-23.7}.$$

Due to (2.10), when event D occurs, we get:

$$\begin{aligned}\Pr(F) &= \Pr(F_1 \cap F_2 \cap F_3) \\ &\approx \Pr(F_1) \cdot \Pr(F_2) \cdot \Pr(F_3) \\ &\approx 2^{-29.7}. \end{aligned} \tag{2.15}$$

Ding et al. assumed that $Z^1_{1(0)} \oplus Z^2_{1(0)} \oplus Z^1_{2(0)} \oplus Z^2_{2(0)}$ is distributed uniformly at random when $D \cap G \cap F$ does not occur. Therefore, assuming that D, G and F are independent events, it is obtained that:

$$\begin{aligned}&\Pr(Z^1_{1(0)} \oplus Z^2_{1(0)} \oplus Z^1_{2(0)} \oplus Z^2_{2(0)} = 0) \\ &\quad = \Pr(D \cap G \cap F) + \frac{1}{2}(1 - \Pr(D \cap G \cap F)) \\ &\quad = \Pr(D) \cdot \Pr(G) \cdot \Pr(F) + \frac{1}{2}(1 - \Pr(D) \cdot \Pr(G) \cdot \Pr(F)) \\ &\quad = \frac{1}{2}(1 + 2^{-74.2}). \end{aligned} \tag{2.16}$$

Based on the bias detected in (2.16), Ding et al. built their alleged related-key distinguishers for Py, Pypy, TPy and TPypy.

2.3.3 Observations on Ding et al.'s Alleged Attacks

RCR-64 and RCR-32

Let us define the following:

$$t_1 := s^2_{1(0)} \oplus s^2_{1(13)} \oplus s^2_{2(0)} \oplus s^2_{2(13)} \oplus \delta_{(13)} \oplus \gamma_{(13)},$$
$$t_2 := c_{1(13)} \oplus c_{2(13)},$$

$$
\begin{aligned}
t_3 &:= Y_1^1[-1]_{(0)} \oplus Y_1^2[-1]_{(0)} \oplus Y_2^1[-1]_{(0)} \oplus Y_2^2[-1]_{(0)} \,, \\
t_4 &:= Y_1^1[P_1^2[208]]_{(0)} \oplus Y_1^2[P_1^3[208]]_{(0)} \oplus Y_2^1[P_2^2[208]]_{(0)} \oplus Y_2^2[P_2^3[208]]_{(0)} \,, \\
\widehat{Z} &:= Z^1_{1(0)} \oplus Z^2_{1(0)} \oplus Z^1_{2(0)} \oplus Z^2_{2(0)} \,.
\end{aligned}
$$

From (2.1)–(2.4), we get:

$$\widehat{Z} = t_1 \oplus t_2 \oplus t_3 \oplus t_4 \,.$$

It is reasonable to assume that t_1 and $\tau := t_2 \oplus t_3 \oplus t_4$ are independent random variables. Let $\Pr(\tau = 0) = w$ and from Sect. 2.3.2, we have:

$$\Pr(t_1 = 0) = \Pr(E_1) = \frac{1}{2} \,.$$

Therefore,

$$
\begin{aligned}
\Pr(t_1 \oplus \tau = 0) &= \Pr(t_1 = 0) \cdot \Pr(\tau = 0) + \Pr(t_1 = 1) \cdot \Pr(\tau = 1) \\
&= \frac{1}{2} w + (1 - \frac{1}{2})(1 - w) \\
&= \frac{1}{2} \,.
\end{aligned} \tag{2.17}
$$

Equation (2.17) proves that $\widehat{Z}$ is not a biased variable. In [146], the authors wrongly assumed that $\Pr(\widehat{Z} = 0) = 0.5$ in the absence of $D \cap G \cap E$ despite the fact that $\widehat{Z}$ equals 0 when the event $D \cap \overline{G} \cap \overline{E}$ occurs (see (2.12)). The probabilities of $\widehat{Z} = 0$ given the occurrence of different combinations of events are listed in Table 2.3. If $\Pr(D) = p$, $\Pr(G) = q$ and $\Pr(E) = r$, assuming that D, G and E are independent events, the probability of $\widehat{Z} = 0$ can be computed as follows:

$$
\begin{aligned}
\Pr(\widehat{Z} = 0) &= p \cdot q \cdot r + p \cdot (1 - q) \cdot (1 - r) + \frac{(1 - p) \cdot (1 - q) \cdot r}{2} \\
&\quad + \frac{(1 - p) \cdot q \cdot (1 - r)}{2} + \frac{(1 - p) \cdot (1 - q) \cdot (1 - r)}{2} \\
&\quad + \frac{(1 - p) \cdot q \cdot r}{2} \,.
\end{aligned} \tag{2.18}
$$

In (2.18), all the events causing $\widehat{Z} = 0$ are taken into consideration and since $r = 0.5$, under the assumption that $s^2_{1(0)} \oplus s^3_{1(0)} \oplus s^2_{2(0)} \oplus s^3_{2(0)}$ is distributed uniformly at random, $\Pr(\widehat{Z} = 0)$ equals 0.5 as computed in (2.17).

Table 2.3: The probabilities p_1, p_2, p_3 and p_4, where $p_1 = \Pr(t_3 = 0 \mid X)$, $p_2 = \Pr(t_4 = 0 \mid X)$, $p_3 = \Pr(t_1 \oplus t_2 = 0 \mid X)$ and $p_4 = \Pr(\widehat{Z} = 0 \mid X)$; X represents different combinations of the events D, G and E

Event X	p_1	p_2	p_3	p_4
$D \cap G \cap E$	1	1	1	1
$D \cap G \cap \overline{E}$	1	1	0	0
$D \cap \overline{G} \cap E$	1	0	1	0
$D \cap \overline{G} \cap \overline{E}$	1	0	0	1
$\overline{D} \cap G \cap E$	0.5	1	1	0.5
$\overline{D} \cap G \cap \overline{E}$	0.5	1	0	0.5
$\overline{D} \cap \overline{G} \cap E$	0.5	0	1	0.5
$\overline{D} \cap \overline{G} \cap \overline{E}$	0.5	0	0	0.5

Py, Pypy, TPy and TPypy

Let us define the following in addition to t_3, t_4 and $\widehat{Z}$ defined in Sect. 2.3.3:

$$\begin{aligned} t_5 &:= s^2_{1(0)} \oplus s^2_{1(d_1)} \oplus s^2_{2(0)} \oplus s^2_{2(d_2)} \oplus \beta_{(d_1)} \oplus \epsilon_{(d_2)} , \\ t_6 &:= e_{1(d_1)} \oplus e_{2(d_2)} , \end{aligned}$$

where e_1, e_2, d_1 and d_2 are defined in Sec. 2.3.2. From (2.1)–(2.4), we find that:

$$\widehat{Z} = t_5 \oplus t_6 \oplus t_3 \oplus t_4 .$$

Let $\Pr(\sigma = 0) = w$, where $\sigma = t_6 \oplus t_3 \oplus t_4$. From Sect. 2.3.2, we have:

$$\Pr(t_5 = 0) = \Pr(F_2) = \frac{1}{2} .$$

Under the assumption that t_5 and σ are independent random variables, we get:

$$\begin{aligned} \Pr(t_5 \oplus \sigma = 0) &= \Pr(t_5 = 0) \cdot \Pr(\sigma = 0) \\ &\quad + \Pr(t_5 = 1) \cdot \Pr(\sigma = 1) \\ &= \frac{1}{2}w + (1 - \frac{1}{2})(1 - w) \\ &= \frac{1}{2}\,. \end{aligned} \tag{2.19}$$

The attacks on Py, Pypy, TPy and TPypy also had errors in the probability calculations, similar to those described in the Sect. 2.3.3, due to which $\Pr(\widehat{Z} = 0)$ as calculated by Ding et al. is different from the correct result in (2.19). Ding et al. wrongly assumed that $\Pr(\widehat{Z} = 0)$ is equal to 0.5 in the absence of $D \cap G \cap F$, in spite of $\widehat{Z}$ being equal to 0 when the event $D \cap \overline{G} \cap \overline{F}$ occurs. The probabilities of $\widehat{Z} = 0$ given the occurrence of different combinations of events are listed in Table 2.4. Under the assumption that D, G and F are independent events, probability of $\widehat{Z} = 0$ can be computed using (2.18), where $\Pr(D) = p$, $\Pr(G) = q$ and $\Pr(F) = r$. Since $r = 0.5$, under the assumption that $s^2_{1(0)} \oplus s^3_{1(0)} \oplus s^2_{2(0)} \oplus s^3_{2(0)}$ is distributed uniformly at random, we get the same result for $\Pr(\widehat{Z} = 0)$ as computed in (2.19).

Table 2.4: The probabilities p_1, p_2, p_3 and p_4, where $p_1 = \Pr(t_3 = 0 \mid Y)$, $p_2 = \Pr(t_4 = 0 \mid Y)$, $p_3 = \Pr(t_5 \oplus t_6 = 0 \mid Y)$ and $p_4 = \Pr(\widehat{Z} = 0 \mid Y)$; Y represents different combinations of the events D, G and F

Event Y	p_1	p_2	p_3	p_4
$D \cap G \cap F$	1	1	1	1
$D \cap G \cap \overline{F}$	1	1	0	0
$D \cap \overline{G} \cap F$	1	0	1	0
$D \cap \overline{G} \cap \overline{F}$	1	0	0	1
$\overline{D} \cap G \cap F$	0.5	1	1	0.5
$\overline{D} \cap G \cap \overline{F}$	0.5	1	0	0.5
$\overline{D} \cap \overline{G} \cap F$	0.5	0	1	0.5
$\overline{D} \cap \overline{G} \cap \overline{F}$	0.5	0	0	0.5

2.4 Conclusions

In this Chapter, we have presented a detailed review of the related-key distinguishing attacks on the stream ciphers Py, Pypy, TPy, TPypy, RCR-64 and RCR-32 proposed in a paper published in the *Journal of Universal Computer Science*. To the best of our knowledge, these are the only claimed attacks on the RCR-64 and RCR-32. The review of the attacks revealed certain flaws in probability calculations which led the authors to visualise biases where they do not exist. In the process, we also discussed about some of the potential flaws in probability-based cryptanalysis.

Chapter 3

Distinguishing Attacks on (Ultra-)lightweight WG Ciphers

3.1 Introduction

Lightweight ciphers. Recently there has been a considerable surge in the popularity of lightweight ciphers due to the advent of Internet of Things. Such ciphers typically use LFSRs, especially in constrained hardware environments. To augment the linear complexity of the keystream, LFSR-based ciphers use balanced nonlinear functions — the resulting keystream generator is called a nonlinear filter generator and well-known examples are the generators of the WG family of ciphers.

The WG family of ciphers. The WG ciphers are based on the WG transformations [166] which are balanced nonlinear filter functions. The possibility of using the WG transformations for cryptographic purposes was first explored by Gong et al. [167]. The transformations are defined over finite fields of orders 2^5, 2^7, 2^8, 2^{16} and 2^{29}; the corresponding ciphers are respectively denoted by WG-5 [168], WG-7 [169], WG-8 [170], WG-16 [171] and WG-29 [172]. Beginning with WG-29 (a well-received entrant to the ECRYPT eSTREAM project [173] designed by Nawaz et al.), the WG ciphers

have been studied extensively, over a period spanning more than a decade. Table 3.1 lists the ciphers and results of their best known security evaluations.

The WG-A and the WG-B,[1] designed by Gong et al., are subfamilies of the WG family and comprise of patented (# US8953784 B2) variants of the WG-8 and WG-16, respectively [142]. WG-A has 3 constituent ultra-lightweight ciphers, each corresponding to a unique *decimation factor* or d. Each of these ciphers supports an 80-bit key and an 80-bit IV. Likewise, WG-B comprises of 31 lightweight ciphers, each using a 128-bit key and a 128-bit IV. The designs of WG-A and WG-B are remarkably simple and the ciphers are well suited for hardware applications. At TechConnect World Innovation Conference 2015, the Waterloo Commercialization Office had exhibited an RFID system for anti-counterfeiting enabled with WG-A [148]. Furthermore, information available through the website of the Waterloo Commercialization Office suggests that WG-B is proposed for securing 4G/5G networks [149]. Consequently, there appear to be good chances for these ciphers to be commercially deployed on a wide scale.

Contributions of this work. Linear distinguishing attacks on WG-A and WG-B families of ciphers are presented in this Chapter. The attacks on WG-A are highly practical, requiring fewer than $2^{29.07}$ keystream samples for nearly guaranteed success, and have been experimentally verified. Our attacks on WG-B, however, are not very practical and require up to $2^{56.84}$ keystream samples to nearly guarantee success. The security claim of the designers of WG-A and WG-B suggests that these ciphers offer better security compared to their predecessors (this is attributed to the decimation factor). Table 3.1 shows that our attacks refute this claim.[2]

Organisation of the Chapter. The remaining Chapter is organised as follows. Section 3.2 describes the ciphers. We present our motivational observations for WG in

[1] We follow this nomenclature to distinguish between the patented and unpatented variants of WG-8 and WG-16.

[2] To facilitate comparisons, we reasonably assume that the success rates of the attacks on WG-16 and WG-B are equal.

Table 3.1: Attacks on the WG family of stream ciphers

Year	Cipher	Key size (in bits)	IV size (in bits)	Type of attack	Requirements	Success rate (%)
2005	WG-29 [a]	80	80	key recovery [174]	$2^{31.3}$ chosen IVs, $O(2^{32.69})$ time	99.95
2007	WG-29 [b]	80, 96, 112, 128 [c]	32, 64 or same as key size [c]	key recovery [175]	$O(2^{45.04})$ data, $O(2^{65.71})$ time	99.99
2012	WG-7	80	81	distinguishing [176]	$O(2^{13.5})$ data	99.99
2012	WG-7	80	81	key recovery [176]	$O(2^{19.38})$ data, $O(2^{27})$ time	100
2014	WG-8	80	80	key recovery [177]	2^{21} chosen IVs, $O(2^{23.29})$ time	99.99
2015	WG-29 [b]	128	128	key recovery [178]	$O(2^{89})$ time, $O(2^{48})$ memory	63.21
2015	WG-5 ($d = 7$)	80	80	key recovery [179]	$O(2^{15})$ data, $O(2^{30})$ time	100
2015	WG-5 ($d = 15$)	80	80	key recovery [179]	$O(2^{15})$ data, $O(2^{30})$ time	100
2015	WG-7	80	81	key recovery [179]	$O(2^{14})$ data, $O(2^{25})$ time	100
2015	WG-8	80	80	key recovery [179]	$O(2^{22})$ data, $O(2^{48})$ time	100
2015	WG-16	128	128	key recovery [179]	$O(2^{63})$ data, $O(2^{106})$ time	100
2016	WG-A	80	80	distinguishing (This Chapter)	up to $2^{33.46}$ bits, less than $O(2^{29.07})$ time	99.99
2016	WG-B	128	128	distinguishing (This Chapter)	up to $2^{61.87}$ bits, less than $O(2^{56.84})$ time	99.99

[a] ECRYPT eSTREAM Phase 1 version
[b] ECRYPT eSTREAM Phase 2 version
[c] attack works for any combination of key size and IV size

Sect. 3.3. The biases in the keystream distribution are computed in Sect. 3.4 and our distinguishing attacks are presented in Sect. 3.5. In Sect. 3.6, we discuss how the remaining members of the WG family fare against linear distinguishing attacks. We conclude in Sect. 3.7.

3.2 Specifications of the Ciphers

3.2.1 WG-A

The ultra-lightweight cipher WG-A uses an 80-bit key K and an 80-bit initialization vector IV. The internal state of WG-A consists of a 20-stage LFSR defined over the finite field $\mathbb{F}_{2^8}$, whose $(k+1)$th stage at time i is denoted by $s_{\text{A}}[i+k]$. The cipher uses an 8-bit nonlinear WG transformation $W_{\text{A}} : \mathbb{F}_{2^8} \to \mathbb{F}_2$. The 256 elements of the finite field $\mathbb{F}_{2^8}$ are generated by the primitive polynomial $R_{\text{A}}(x) = x^8 + x^4 + x^3 + x^2 + 1$ over $\mathbb{F}_2$. Let ω be a root of the primitive polynomial. Any element of $\mathbb{F}_{2^8}$ can be represented as an 8-bit binary vector $(a_7 a_6 \dots a_0)$ corresponding to $a_7\omega^7 + a_6\omega^6 + \cdots + a_0$. The feedback polynomial of the LFSR is given by $l_{\text{A}}(x) = x^{20} + x^{11} + x^9 + \omega^{38}$, where ω^{38} corresponds to the binary vector (10010100). The WG transformation, whose input is the 20th LFSR stage, is comprised of a permutation P_{A} and a trace function T_{A}. The permutation takes an 8-bit input x and outputs $q_{\text{A}}(x \boxplus_8 1) \boxplus_8 1$, where $q_{\text{A}}(x) = x \boxplus_8 x^9 \boxplus_8 x^{57} \boxplus_8 x^{73} \boxplus_8 x^{71}$. Likewise, the trace function takes an 8-bit input x and outputs $x \boxplus_8 x^2 \boxplus_8 x^{2^2} \boxplus_8 \cdots \boxplus_8 x^{2^7}$. The trace function acts on $P_{\text{A}}(x)$ to yield $W_{\text{A}}(x)$ as $T_{\text{A}}(P_{\text{A}}(x))$. For better security of the cipher, the designers propose to have x^d instead of x, where d is the decimation factor, as the input to the WG transformation [142]. Although the value of d is not mentioned in the patent document, Mandal et al. identify three values (13, 19 and 61) as "optimal" choices [180]. These values, they argue, impart the best cryptographic properties such as maximum algebraic degree, maximum algebraic immunity, largest possible nonlinearity and smallest possible additive correlation to the output of W_{A}. Denote by WG-A$_d$ the cipher corresponding to d. Then, WG-A$_d$ operates in two phases: initialization and keystream generation (see Algorithms 4 and 5). The keystream generation process of WG-A$_d$ is shown in Figure 3.1.

Algorithm 4 Initialization of WG-A$_d$

Require: A key and an IV

Ensure: An LFSR $\{s_{\text{A}}[40+19], s_{\text{A}}[40+18], \ldots, s_{\text{A}}[40+0]\}$

1: **for** i = 0 to 9 **do**

2: $s_{\text{A}}[2i] = (IV_{(8i+3)}, IV_{(8i+2)}, IV_{(8i+1)}, IV_{(8i)}, K_{(8i+3)}, K_{(8i+2)}, K_{(8i+1)}, K_{(8i)})$;

3: $s_{\text{A}}[2i+1] = (IV_{(8i+7)}, IV_{(8i+6)}, IV_{(8i+5)}, IV_{(8i+4)}, K_{(8i+7)}, K_{(8i+6)}, K_{(8i+5)}, K_{(8i+4)})$;

4: **endfor**

5: **for** i = 0 to 39 **do**

6: $s_{\text{A}}[i+20] = s_{\text{A}}[i+11] \oplus s_{\text{A}}[i+9] \oplus (s_{\text{A}}[i] \boxdot_8 \omega^{38}) \oplus P_{\text{A}}((s_{\text{A}}[i+19])^d)$;

7: **endfor**

Algorithm 5 Keystream generator of WG-A$_d$

Require: An LFSR $\{s_{\text{A}}[19], s_{\text{A}}[18], \ldots, s_{\text{A}}[0]\}$

Ensure: A pseudorandom output z_{A}

1: $i = 0$;

2: **do** until enough keystream is generated

3: $z_{\text{A}(i)} = W_{\text{A}}((s_{\text{A}}[i+19])^d)$;

4: $s_{\text{A}}[i+20] = s_{\text{A}}[i+11] \oplus s_{\text{A}}[i+9] \oplus (s_{\text{A}}[i] \boxdot_8 \omega^{38})$;

5: $i \leftarrow i+1$;

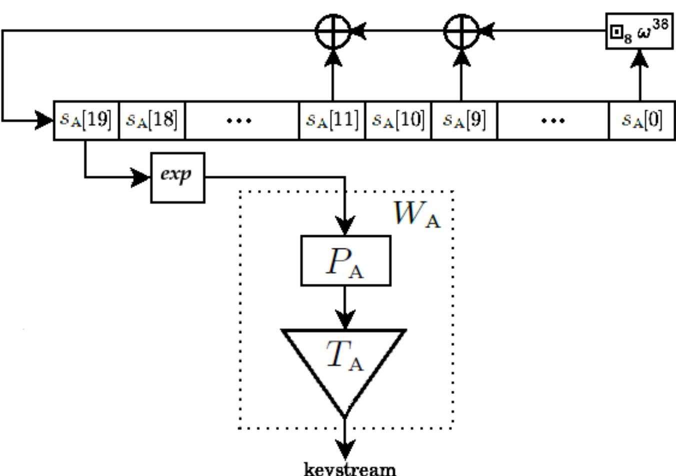


Figure 3.1: Keystream generation of WG-A$_d$; exp computes $(s_{\text{A}}[19])^d$

3.2.2 WG-B

The structure of the lightweight cipher WG-B is very similar to that of WG-A. Here, the key K and the initialization vector IV are of length 128 bits each. The LFSR has 32 stages (which are denoted by $s_{\text{B}}[\cdot]$) and is defined over the finite field $\mathbb{F}_{2^{16}}$. The elements of $\mathbb{F}_{2^{16}}$ are generated by β, a primitive root of the polynomial $R_{\text{B}}(x) = x^{16} + x^5 + x^3 + x^2 + 1$ over $\mathbb{F}_2$. The feedback polynomial is given by $l_{\text{B}}(x) = x^{32} + x^{13} + x^3 + \beta^2 + 1$. The permutation is of 16-bit values and is given by $P_{\text{B}}(x) = q_{\text{B}}(x \boxplus_{16} 1) \boxplus_{16} 1$, where $q_{\text{B}}(x) = x \boxplus_{16} x^{2049} \boxplus_{16} x^{2111} \boxplus_{16} x^{2113} \boxplus_{16} x^{63552}$. The trace function and the nonlinear filter function are from $\mathbb{F}_{2^{16}}$ to $\mathbb{F}_2$ and are given by:

$$\begin{aligned} T_{\text{B}}(x) &= x \boxplus_{16} x^2 \boxplus_{16} x^{2^2} \boxplus_{16} \cdots \boxplus_{16} x^{2^{15}}, \\ W_{\text{B}}(x) &= T_{\text{B}}(P_{\text{B}}(x)). \end{aligned}$$

As for WG-A, optimal decimation factors (31 in total) have also been identified for WG-B in [180] and WG-B$_d$ refers to the cipher corresponding to d. It operates in two phases: initialization and keystream generation (see Algorithms 6 and 7). Figure 3.2 shows the keystream generation of WG-B$_d$.

Algorithm 6 Initialization of WG-B$_d$

Require: A key and an IV

Ensure: An LFSR $\{s_{\text{B}}[64+31], s_{\text{B}}[64+30], \ldots, s_{\text{B}}[64+0]\}$

1: **for** i = 0 to 15 **do**
2: $s_{\text{B}}[i] = (IV_{(8i+7)}, IV_{(8i+6)}, \ldots, IV_{(8i)}, K_{(8i+7)}, K_{(8i+6)}, \ldots, K_{(8i)})$;
3: $s_{\text{B}}[i+16] = s_{\text{B}}[i]$;
4: **endfor**
5: **for** i = 0 to 63 **do**
6: $s_{\text{B}}[i+32] = s_{\text{B}}[i+13] \oplus s_{\text{B}}[i+3] \oplus (s_{\text{B}}[i] \boxdot_{16} (\beta^2+1)) \oplus P_{\text{B}}((s_{\text{B}}[i+31])^d)$;
7: **endfor**

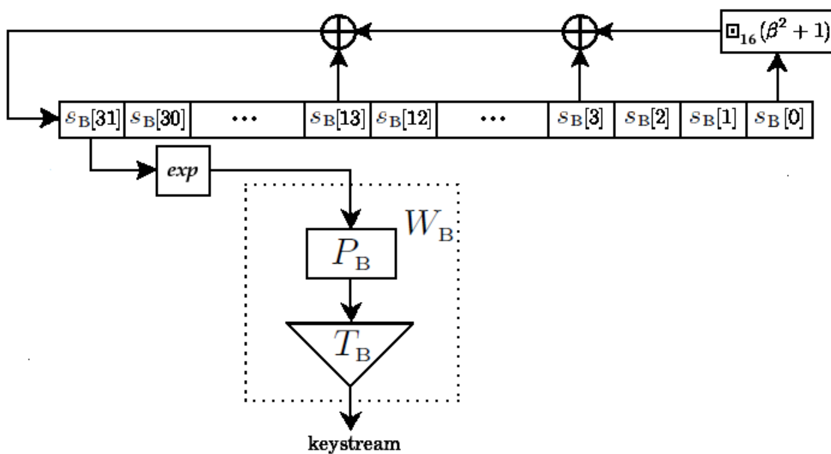


Figure 3.2: Keystream generation of WG-B$_d$; exp computes $(s_B[31])^d$

Algorithm 7 Keystream generator of WG-B$_d$

Require: An LFSR $\{s_B[31], s_B[30], \ldots, s_B[0]\}$
Ensure: A pseudorandom output z_B

1: $i = 0$;
2: **do** until enough keystream is generated
3: $z_{B(i)} = W_B((s_B[i+31])^d)$;
4: $s_B[i+32] = s_B[i+13] \oplus s_B[i+3] \oplus (s_B[i] \boxdot_{16} (\beta^2+1))$;
5: $i \leftarrow i+1$;

3.3 Motivational Observation

The sizes of the inputs to W_A and W_B render an exhaustive search over the input space feasible. Performing the search, we detect d independent input-output correlations in the WG transformations. The bitwise correlation probabilities for $W_A(x^{61})$ and $W_B(x^{157})$ are listed in Tables 3.2(a) and 3.3(a), respectively. Similarly, the probabilities that $W_A(x^{61}) = (x \boxdot_8 \omega^{38})_{(i)}$ and $W_B(x^{157}) = (x \boxdot_{16} (\beta^2+1))_{(i)}$, for several values of i, are listed in Tables 3.2(b) and 3.3(b), respectively. All the results presented in this Chapter are based on the polynomial basis representation of the field elements.

Tables 3.2(a), 3.2(b), 3.3(a) and 3.3(b) suggest that the WG transformations can be

Table 3.2: WG: Probabilities that (a) $W_{\text{A}}(x^{61}) = x_{(i)}$ and (b) $W_{\text{A}}(x^{61}) = (x \boxdot_8 \omega^{38})_{(i)}$, for several values of i

(a)

i	Pr $\left(W_{\text{A}}(x^{61}) = x_{(i)}\right)$
0	$0.5 + 2^{-4.68}$
1	$0.5 + 2^{-6.00}$
2	$0.5 + 2^{-5.00}$
3	$0.5 - 2^{-6.00}$
4	$0.5 - 2^{-5.00}$
5	$0.5 - 2^{-4.00}$
6	$0.5 + 2^{-6.00}$
7	$0.5 + 2^{-6.00}$

(b)

i	Pr $\left(W_{\text{A}}(x^{61}) = (x \boxdot_8 \omega^{38})_{(i)}\right)$
0	$0.5 - 2^{-5.00}$
1	$0.5 + 2^{-4.00}$
2	$0.5 - 2^{-7.00}$
3	$0.5 + 2^{-4.00}$
4	$0.5 + 2^{-7.00}$
5	$0.5 - 2^{-6.00}$
6	$0.5 + 2^{-6.00}$
7	$0.5 - 2^{-3.83}$

Table 3.3: WG: Probabilities that (a) $W_{\text{B}}(x^{157}) = x_{(i)}$ and (b) $W_{\text{B}}(x^{157}) = (x \boxdot_{16} (\beta^2 + 1))_{(i)}$, for several values of i

(a)

i	Pr $\left(W_{\text{B}}(x^{157}) = x_{(i)}\right)$
0	$0.5 + 2^{-9.79}$
1	$0.5 + 2^{-12.4}$
2	$0.5 + 2^{-10.8}$
3	$0.5 + 2^{-8.96}$
4	$0.5 + 2^{-10.2}$
5	$0.5 - 2^{-13.0}$
6	$0.5 + 2^{-14.0}$
7	$0.5 + 2^{-10.4}$
8	$0.5 - 2^{-9.75}$
10	$0.5 - 2^{-9.48}$
11	$0.5 + 2^{-9.19}$
12	$0.5 + 2^{-11.7}$
13	$0.5 - 2^{-9.00}$
14	$0.5 - 2^{-11.0}$
15	$0.5 + 2^{-10.5}$

(b)

i	Pr $\left(W_{\text{B}}(x^{157}) = (x \boxdot_{16} (\beta^2 + 1))_{(i)}\right)$
0	$0.5 - 2^{-10.8}$
1	$0.5 + 2^{-10.1}$
2	$0.5 - 2^{-9.17}$
3	$0.5 - 2^{-9.48}$
4	$0.5 - 2^{-9.48}$
5	$0.5 + 2^{-13.0}$
6	$0.5 + 2^{-8.51}$
7	$0.5 - 2^{-13.0}$
8	$0.5 - 2^{-7.91}$
10	$0.5 - 2^{-13.0}$
11	$0.5 + 2^{-13.0}$
12	$0.5 + 2^{-13.0}$
13	$0.5 - 2^{-9.42}$
14	$0.5 + 2^{-10.1}$
15	$0.5 + 2^{-10.1}$

linearly approximated — using this, we state and prove Theorems 1 and 2.

Theorem 1. *If the conditions*

$$
\begin{aligned}
W_{A}(x^{d}) &= x_{(i)} , && (3.1)\\
W_{A}(x^{d}) &= (x \boxdot_{8} \omega^{38})_{(i)} , && (3.2)
\end{aligned}
$$

are satisfied for any $i \in \{0, 1, \ldots, 7\}$*, then the keystream of WG-A*$_d$ *satisfies* $z_{A(t+1)} \oplus z_{A(t-8)} \oplus z_{A(t-10)} \oplus z_{A(t-19)} = 0$ *for* $t \geq 19$.

Proof. The recurrence relation of the constituent LFSR of WG-A$_d$ can be deduced from its feedback polynomial as:

$$s_{\mathrm{A}}[t+20] = s_{\mathrm{A}}[t+11] \oplus s_{\mathrm{A}}[t+9] \oplus (s_{\mathrm{A}}[t] \boxdot_{8} \omega^{38}) , \text{ for } t \geq 0. \quad (3.3)$$

From (3.3), we get:

$$s_{\mathrm{A}}[t+20]_{(i)} = s_{\mathrm{A}}[t+11]_{(i)} \oplus s_{\mathrm{A}}[t+9]_{(i)} \oplus (s_{\mathrm{A}}[t] \boxdot_{8} \omega^{38})_{(i)} , \quad (3.4)$$

for any $i \in \{0, 1, \ldots, 7\}$, $t \geq 19$. Substituting (3.1) and (3.2) in (3.4) yields:

$$W_{\mathrm{A}}((s_{\mathrm{A}}[t+20])^{d}) = W_{\mathrm{A}}((s_{\mathrm{A}}[t+11])^{d}) \oplus W_{\mathrm{A}}((s_{\mathrm{A}}[t+9])^{d}) \oplus W_{\mathrm{A}}((s_{\mathrm{A}}[t])^{d}) , \quad (3.5)$$

for $t \geq 19$. Since $z_{\mathrm{A}(t)} = W_{\mathrm{A}}((s_{\mathrm{A}}[t+19])^{d})$, (3.5) reduces to:

$$z_{\mathrm{A}(t+1)} \oplus z_{\mathrm{A}(t-8)} \oplus z_{\mathrm{A}(t-10)} \oplus z_{\mathrm{A}(t-19)} = 0 , \text{ for } t \geq 19.$$

The proof completes. □

Theorem 2. *If the conditions*

$$W_B(x^d) = x_{(i)}\,, \tag{3.6}$$

$$W_B(x^d) = (x \boxdot_{16} (\beta^2+1))_{(i)}\,, \tag{3.7}$$

are satisfied for any $i \in \{0,1,\ldots,15\}$*, then the keystream of WG-B*$_d$ *satisfies* $z_{B(t+1)} \oplus z_{B(t-18)} \oplus z_{B(t-28)} \oplus z_{B(t-31)} = 0$ *for* $t \geq 31$.

Proof. The LFSR of WG-B$_d$ is given by the following recursion:

$$s_{\mathrm{B}}[t+32] = s_{\mathrm{B}}[t+13] \oplus s_{\mathrm{B}}[t+3] \oplus (s_{\mathrm{B}}[t] \boxdot_{16} (\beta^2+1))\,, \text{ for } t \geq 0. \tag{3.8}$$

For the ith bit, (3.8) becomes:

$$s_{\mathrm{B}}[t+32]_{(i)} = s_{\mathrm{B}}[t+13]_{(i)} \oplus s_{\mathrm{B}}[t+3]_{(i)} \oplus (s_{\mathrm{B}}[t] \boxdot_{16} (\beta^2+1))_{(i)}\,, \tag{3.9}$$

for any $i \in \{0,1,\ldots,15\}$, $t \geq 31$. Substituting (3.6) and (3.7) in (3.9), we get:

$$W_{\mathrm{B}}((s_{\mathrm{B}}[t+32])^d) = W_{\mathrm{B}}((s_{\mathrm{B}}[t+13])^d) \oplus W_{\mathrm{B}}((s_{\mathrm{B}}[t+3])^d) \oplus W_{\mathrm{B}}((s_{\mathrm{B}}[t])^d)\,, \text{ for } t \geq 31. \tag{3.10}$$

Since $z_{\mathrm{B}(t)} = W_{\mathrm{B}}((s_{\mathrm{B}}[t+31])^d)$, (3.10) reduces to:

$$z_{\mathrm{B}(t+1)} \oplus z_{\mathrm{B}(t-18)} \oplus z_{\mathrm{B}(t-28)} \oplus z_{\mathrm{B}(t-31)} = 0\,, \text{ for } t \geq 31.$$

This completes the proof. □

3.4 Bias Estimation

Using the results of Sect. 3.3, we proceed to compute $\Pr(\hat{z}_{\text{A}} = 0)$ and $\Pr(\hat{z}_{\text{B}} = 0)$, where

$$\begin{aligned} \hat{z}_{\text{A}} &= z_{\text{A}(t+1)} \oplus z_{\text{A}(t-8)} \oplus z_{\text{A}(t-10)} \oplus z_{\text{A}(t-19)} \,,\; t \geq 19 \,, \\ \hat{z}_{\text{B}} &= z_{\text{B}(t+1)} \oplus z_{\text{B}(t-18)} \oplus z_{\text{B}(t-28)} \oplus z_{\text{B}(t-31)} \,,\; t \geq 31 \,. \end{aligned}$$

3.4.1 Biases in the Keystream of WG-A$_{61}$

Let us define the Boolean variables Y_1, Y_2, Y_3, Y_4 and Y_5 as follows:

$$\begin{aligned} Y_1 &= W_{\text{A}}((s_{\text{A}}[t+20])^d) \oplus s_{\text{A}}[t+20]_{(i)} \,, \\ Y_2 &= W_{\text{A}}((s_{\text{A}}[t+11])^d) \oplus s_{\text{A}}[t+11]_{(i)} \,, \\ Y_3 &= W_{\text{A}}((s_{\text{A}}[t+9])^d) \oplus s_{\text{A}}[t+9]_{(i)} \,, \\ Y_4 &= W_{\text{A}}((s_{\text{A}}[t])^d) \oplus (s_{\text{A}}[t] \boxdot_8 \omega^{38})_{(i)} \,, \\ Y_5 &= z_{\text{A}(t+1)} \oplus z_{\text{A}(t-8)} \oplus z_{\text{A}(t-10)} \oplus z_{\text{A}(t-19)} \,, \end{aligned}$$

for any $i \in \{0, 1, \ldots, 7\}$, $t \geq 19$. From Theorem 1, we construct the Boolean truth table given in Table 3.4.

Let $\Pr\left(W_{\text{A}}(x^d) = x_{(i)}\right) = p_i$ and $\Pr\left(W_{\text{A}}(x^d) = (x \boxdot_8 \omega^{38})_{(i)}\right) = q_i$. We get:

$$\Pr(Y_1 = 0) = \Pr(Y_2 = 0) = \Pr(Y_3 = 0) = p_i \,, \tag{3.11}$$

$$\Pr(Y_4 = 0) = q_i \,, \tag{3.12}$$

$$\Pr(Y_5 = 0) = \Pr(\hat{z}_{\text{A}} = 0) \,. \tag{3.13}$$

We assume that the events corresponding to Y_1, Y_2, Y_3 and Y_4 are independent and the events corresponding to the rows of the truth table given in Table 3.4 are mutually ex-

Table 3.4: Truth table that satisfies the relation between the Boolean variables Y_1, Y_2, Y_3, Y_4 and Y_5

Y_1	Y_2	Y_3	Y_4		Y_5
0	0	0	0		0
0	0	0	1		1
0	0	1	0		1
0	0	1	1		0
0	1	0	0		1
0	1	0	1		0
0	1	1	0		0
0	1	1	1		1
1	0	0	0		1
1	0	0	1		0
1	0	1	0		0
1	0	1	1		1
1	1	0	0		0
1	1	0	1		1
1	1	1	0		1
1	1	1	1		0

clusive. Then, the truth table given in Table 3.4 and (3.11)–(3.13) yield:

$$\Pr(\hat{z}_{\text{A}} = 0) = p_i^3 q_i + 3(1-p_i)^2 p_i q_i + 3p_i^2(1-p_i)(1-q_i) + (1-p_i)^3(1-q_i)\,. \quad (3.14)$$

A simple application of the Matsui's Piling-up Lemma [115] also leads to (3.14).

The value of $\Pr(\hat{z}_{\text{A}} = 0)$ varies with i and the probability for which $|\Pr(\hat{z}_{\text{A}} = 0) - 0.5|$ is maximised is considered as its best estimation. The values of p_i and q_i for $d = 61$ are listed in Tables 3.2(a) and 3.2(b), respectively. Among the available choices of i, the following linear approximations, corresponding to $i = 5$, maximise the bias.

$$\begin{aligned} W_{\text{A}}(x^{61}) &\approx x_{(5)}\,, \\ W_{\text{A}}(x^{61}) &\approx (x \boxdot_8 \omega^{38})_{(5)}\,. \end{aligned}$$

Since $\Pr\left(W_{\text{A}}(x^{61}) = x_{(5)}\right) = 0.5 - 2^{-4}$ and $\Pr\left(W_{\text{A}}(x^{61}) = (x \boxdot_8 \omega^{38})_{(5)}\right) = 0.5 - 2^{-6}$,

the best estimation of the probability that $\hat{z}_{\text{A}}$ equals zero is $0.5 + 2^{-15}$.

3.4.2 Biases in the Keystream of WG-B$_{157}$

If $\Pr\left(W_{\text{B}}(x^d) = x_{(i)}\right) = p_i$ and $\Pr\left(W_{\text{B}}(x^d) = (x \boxdot_{16} (\beta^2+1))_{(i)}\right) = q_i$ then $\Pr(\hat{z}_{\text{B}} = 0)$ is again given by the RHS of (3.14). The values of p_i and q_i for $d = 157$ are listed in Tables 3.3(a) and 3.3(b), respectively. If the keystream bits are generated by WG-B$_{157}$, the probability that $\hat{z}_{\text{B}}$ equals zero is estimated to be $0.5 - 2^{-33.36}$ based on the following linear approximations:

$$\begin{aligned} W_{\text{B}}(x^{157}) &\approx x_{(3)}\,, \\ W_{\text{B}}(x^{157}) &\approx (x \boxdot_{16} (\beta^2+1))_{(3)}\,. \end{aligned}$$

3.4.3 Improvements to the Bias Estimations

The probabilities $\Pr(\hat{z}_{\text{A}} = 0)$ and $\Pr(\hat{z}_{\text{B}} = 0)$ of Sections 3.4.1 and 3.4.2 were calculated, in each case, with one of the input bits of the WG transformations. Since multiple input bits of W_{A} and W_{B} are correlated to the corresponding output bits, we explore the possibility to obtain a better estimation of the probabilities by combining input bits. The probabilities $\Pr\left(W_{\text{A}}(x^{61}) = \bigoplus_{i\in S} x_{(i)}\right)$ and $\Pr\left(W_{\text{A}}(x^{61}) = \bigoplus_{i\in S} (x \boxdot_8 \omega^{38})_{(i)}\right)$, where $S \subseteq \{0,1,\ldots,7\}$ for WG-A$_{61}$ and equivalently for WG-B$_{157}$, were experimentally calculated with arbitrary choices of S. The correlation probabilities which further improved the estimated keystream biases of WG-A$_{61}$ and WG-B$_{157}$ are given below.

$$\Pr(W_{\text{A}}(x^{61}) = \bigoplus_{i\in S_1} x_{(i)}) = 0.5 - 2^{-3.83}\,, \tag{3.15}$$

$$\Pr(W_{\text{A}}(x^{61}) = \bigoplus_{i\in S_1} (x \boxdot_8 \omega^{38})_{(i)}) = 0.5 + 2^{-4}\,, \tag{3.16}$$

$$\Pr(W_{\text{B}}(x^{157}) = \bigoplus_{i\in S_2} x_{(i)}) = 0.5 + 2^{-6.83}\,, \tag{3.17}$$

$$\Pr(W_{\text{B}}(x^{157}) = \bigoplus_{i\in S_2} (x \boxdot_{16} (\beta^2+1))_{(i)}) = 0.5 - 2^{-8}\,, \tag{3.18}$$

Table 3.5: Probabilities that (a) $\hat{z}_{\text{A}} = 0$ and (b) $\hat{z}_{\text{B}} = 0$, for each of the WG-A$_d$ and WG-B$_d$ ciphers

(a) WG-A$_d$

d	$\Pr(\hat{z}_{\text{A}} = 0)$
13	$0.5 - 2^{-9.25}$
19	$0.5 + 2^{-12.68}$
61	$0.5 - 2^{-12.49}$

(b) WG-B$_d$

d	$\Pr(\hat{z}_{\text{B}} = 0)$		d	$\Pr(\hat{z}_{\text{B}} = 0)$
157	$0.5 - 2^{-25.49}$		3419	$0.5 - 2^{-26.48}$
409	$0.5 - 2^{-25.74}$		3449	$0.5 - 2^{-25.62}$
451	$0.5 - 2^{-25.21}$		3581	$0.5 - 2^{-25.59}$
469	$0.5 + 2^{-26.52}$		4411	$0.5 - 2^{-26.19}$
1057	$0.5 + 2^{-25.36}$		4681	$0.5 + 2^{-26.17}$
1187	$0.5 - 2^{-26.20}$		4789	$0.5 - 2^{-26.56}$
1327	$0.5 - 2^{-26.43}$		5213	$0.5 + 2^{-26.23}$
1393	$0.5 + 2^{-25.55}$		6043	$0.5 + 2^{-26.03}$
1397	$0.5 - 2^{-25.29}$		7673	$0.5 - 2^{-25.04}$
1771	$0.5 + 2^{-25.51}$		7771	$0.5 - 2^{-26.19}$
1933	$0.5 + 2^{-25.38}$		10651	$0.5 - 2^{-26.32}$
2137	$0.5 - 2^{-24.01}$		10667	$0.5 - 2^{-25.38}$
2251	$0.5 + 2^{-24.90}$		13631	$0.5 - 2^{-25.99}$
2473	$0.5 - 2^{-26.09}$		14327	$0.5 - 2^{-25.66}$
2741	$0.5 - 2^{-25.07}$		32767	$0.5 + 2^{-26.57}$
3223	$0.5 - 2^{-25.86}$			

where $S_1 = \{0, 3, 5, 6\}$ and $S_2 = \{1, 3, 5, 6, 8, 9, 14, 15\}$. Assigning the probabilities of (3.15) and (3.16) respectively for p_i and q_i, (3.14) yields $\Pr(\hat{z}_{\text{A}} = 0) = 0.5 - 2^{-12.49}$ for WG-A$_{61}$. Similarly, the assignments $p_i = 0.5 + 2^{-6.83}$ and $q_i = 0.5 - 2^{-8}$ yield $\Pr(\hat{z}_{\text{B}} = 0) = 0.5 - 2^{-25.49}$ for WG-B$_{157}$. The WG-A and WG-B families are restricted to ciphers corresponding to the optimal d; the values of $\Pr(\hat{z}_{\text{A}} = 0)$ and $\Pr(\hat{z}_{\text{B}} = 0)$ for all these ciphers are listed in Tables 3.5(a) and 3.5(b), respectively.

3.5 Attack Complexities

In this Section, we compute the complexities of our distinguishing attacks on WG-A and WG-B using the results of Sect. 3.4. Let n denote the number of $\hat{z}_{\text{A}}$'s available to the attacker, D' denote the distribution of $Z := \sum_n \hat{z}_{\text{A}}$, $p' = \Pr(\hat{z}_{\text{A}} = 0)$, D denote the distribution of Z given that WG-A$_{61}$ is an ideal cipher and $p = 0.5$. If the $\hat{z}_{\text{A}}$'s are independent and identically distributed random variables (i.i.d.), then Z has a binomial

distribution. The means (μ', μ) and standard deviations (σ', σ) of the distributions D', D are given by: $\mu' = np'$, $\mu = np$, $\sigma' = \sqrt{np'(1-p')}$ and $\sigma = \sqrt{np(1-p)}$.

If n is large (a commonly used rule of thumb is that $np > 5$ and $n(1-p) > 5$), one can approximate each binomial distribution with the normal distribution with the same mean and standard deviation. Let $\mathcal{N}(\mu,\, \sigma^2)$ and $\mathcal{N}(\mu',\, \sigma'^2)$ be the normal distributions corresponding to D and D', respectively. If $\mu > \mu'$ (resp. $\mu < \mu'$) and $\mu - 3.62\sigma > \mu' + 3.62\sigma'$ (resp. $\mu + 3.62\sigma < \mu' - 3.62\sigma'$), the attacker can confirm that $\hat{z}_{\text{A}}$'s follow the distribution $\mathcal{N}(\mu',\, \sigma'^2)$ with 0.9999 confidence level.[3] Given this, if $|\mu' - \mu| > 3.62(\sigma' + \sigma) \Rightarrow n > 13.1/(p' - 0.5)^2 = 2^{28.69}$, the cipher WG-$\text{A}_{61}$ can be distinguished from an ideal cipher with 99.99% success rate and 0.01% false positive rate. Similarly, $2^{54.69}$ keystream samples ($\hat{z}_{\text{B}}$) are required to distinguish WG-B_{157} from an ideal cipher, with a success probability of 0.9999. In order to generate the keystream samples $\hat{z}_{\text{A}}$ and $\hat{z}_{\text{B}}$, in the worst case, the attacker collects 21 keystream bits and 33 keystream bits, respectively per (K, IV) pair (but actually 4 bits will suffice). The data complexities of our distinguishing attacks on the other members of the WG-A and WG-B families are respectively listed in Tables 3.6(a) and 3.6(b)—in each case, the success rate is 99.99%.

3.5.1 Experimental Verification

From Table 3.6(a), it is clear that the distinguishing attacks on WG-A are of practical complexities. In order to verify our analysis, the attacks on WG-A_{13}, WG-A_{19} and WG-A_{61} were simulated. In each case, the keystream bits, as per the data requirement given in Table 3.6(a), were generated from 2^{20} (K, IV) pairs chosen uniformly at random with each pair generating 10^3 keystream bits and the required probability was computed.[4] This process was repeated 10^4 times. The mean probabilities for WG-A_{13}, WG-A_{19} and WG-A_{61} were found to be $0.5 - 2^{-9.40}$, $0.5 + 2^{-15.10}$ and $0.5 - 2^{-12.51}$, respectively.

[3] The cumulative distribution function of the normal distribution gives the value 0.9999 at $\mu + 3.62\sigma$.

[4] In order to compute the time complexity of our distinguishing attacks we assume that the attacker collects one keystream sample per (K, IV) pair. It is reasonable to expect the results of our simulations to agree with simulations performed with 2^{30} (K, IV) pairs chosen uniformly at random and one keystream sample per (K, IV) pair.

Table 3.6: Data requirements of our attacks (corresponding to 0.9999 success probability) on the WG-A_d and WG-B_d ciphers

(a) WG-A_d

d	$\log_2(\#samples)$
13	22.20
19	29.07
61	28.69

(b) WG-B_d

d	$\log_2(\#samples)$		d	$\log_2(\#samples)$
157	54.69		3419	56.66
409	55.19		3449	54.95
451	54.13		3581	54.89
469	56.76		4411	56.09
1057	54.43		4681	56.06
1187	56.11		4789	56.82
1327	56.57		5213	56.16
1393	54.8		6043	55.78
1397	54.3		7673	53.78
1771	54.73		7771	56.09
1933	54.47		10651	56.35
2137	51.74		10667	54.47
2251	53.51		13631	55.70
2473	55.9		14327	55.03
2741	53.85		32767	56.84
3223	55.43			

The theoretical and experimental results for WG-A_{13} and WG-A_{61} agree very well; the reason why the agreement is not as pronounced in the case of WG-A_{19} is being investigated.

3.6 Discussion

In [176], Orumiehchiha et al. report a linear distinguishing attack, that is similar to our attacks, on WG-7. Our investigations, in fact, show that every member of the WG family, with the sole exception of the WG-29, is vulnerable to such linear attacks. The low correlation immunity and the low resilience of the WG transformations allow us to identify linear approximations of the kind provided in Sect. 3.4.3.

For WG-29, the input size of the WG transformation is too large to perform an exhaustive search over the input space using a general purpose computer. Nawaz et al. report that it is 1-order resilient and can be approximated by linear functions [172].

Therefore, the possibility of the existence of a set, similar to the set S of Sect. 3.4.3, that renders WG-29 vulnerable to linear distinguishing attacks cannot be eliminated.

3.7 Conclusions

In this Chapter, we presented distinguishing attacks on the stream ciphers WG-A and WG-B. To nearly guarantee success, the attacks require between $2^{22.20}$ and $2^{29.07}$ keystream samples for WG-A, and fewer than $2^{56.84}$ keystream samples for WG-B. Let $T_{\text{A},ini}$ and $T_{\text{A},kga}$ respectively denote the run-times of the initialization algorithm and keystream generation algorithm of WG-A. Then, assuming that one sample is collected per (K, IV) pair, our attacks on the WG-A ciphers each requires at the most $2^{29.07}(T_{\text{A},ini} + 21 \cdot T_{\text{A},kga})$ time.[5] Likewise, our attacks on the WG-B ciphers each requires at the most $2^{56.84}(T_{\text{B},ini} + 33 \cdot T_{\text{B},kga})$ time, where $T_{\text{B},ini}$ and $T_{\text{B},kga}$ are the respective run-times of the initialization algorithm and keystream generation algorithm of WG-B. For a success rate of 60%, the attacks on WG-A and WG-B respectively require not more than $2^{21.47}$ and $2^{49.24}$ keystream samples, and equivalent time. To the best of our knowledge, these are the first attacks on the WG-A as well as the WG-B.

The low nonlinearity and the low correlation immunity of the WG transformations appear to be the main causes of these attacks. As pointed out in [170] and [181], increasing the number of tap positions of the LFSRs used in the WG ciphers may increase the complexity of the distinguishing attacks. For instance, if there are 9 tap positions instead of 3 in the LFSR of WG-A_{61}, our attack will require $2^{62.65}$ keystream samples for a success rate of 99.99%. Nevertheless, to preclude these attacks, we recommend using filter functions having good correlation immunity.

[5] An inherent assumption is that the decimation factor has no bearing on the run-time of the cipher.

Chapter 4

Side-Channel Analysis of SPECK

4.1 Introduction

The SPECK family of ciphers. SPECK is a family of lightweight block ciphers designed by Beaulieu et al. of the NSA "as an aid for securing applications in very constrained environments where AES may not be suitable" [96]. The design principle to use basic operations such as modular addition, bitwise XOR and circular shifts has made them remarkably simple and highly flexible across the platforms. SPECK has 10 constituent ciphers, each corresponding to a unique combination of key size and block size. The key size varies from 64 bits to 256 bits whereas the block size ranges from 32 bits to 128 bits. The use of modular addition operation for nonlinearity has made SPECK well optimised for software applications. The ISO/IEC specifies a crypto suite for SPECK for their air interfaces standards for RFID devices [143]. To be used on low-end Android Go devices, Google added SPECK to Linux Kernel 4.17 [182] but eventually it was removed from Linux Kernel 4.20 and subsequent versions [183, 184].

Related works. Given the possibility for widespread deployment in lightweight applications, SPECK has been studied extensively during the recent years. In 2014, Abed et al. presented differential attacks on 10, 12, 15, 15 and 16 rounds of SPECK ciphers

with block sizes 32, 48, 64, 96, and 128 bits, respectively [185]. They also reported rectangle attacks which could work on 11 and 18 rounds of SPECK ciphers with 64- and 256-bit keys, respectively [185]. In the same year, Biryukov et al. also proposed differential cryptanalysis of reduced-round SPECK which could be used to attack 16 rounds of SPECK ciphers with a block size of 64 bits [186]. Later, Dinur published an improved differential cryptanalysis of SPECK which increased the number of rounds that can be attacked by 1, 2, or 3, for 9 out of 10 reduced-round members of the family, while significantly improving the complexity of the previous best-known attack on the remaining reduced-round member [187].

The first known attacks on full SPECK ciphers were by Tupsamudre et al. who presented a differential fault analysis on SPECK which recovers the n-bit subkey of the final round using $n/3$ bit faults on an average [188]. Tupsamudre et al.'s differential fault attacks were further improved by Huo et al., whose attacks required a more practical random fault model and lesser number of fault injections compared to the earlier attacks [189]. In 2015, Yuan et al. reported the first known linear cryptanalysis on reduced-round SPECK ciphers [190]. In 2016, Fu et al. proposed differential attacks on reduced-round SPECK ciphers with block sizes 48, 64, 96 and 128 bits, which were better than the earlier differential attacks in terms of the number of rounds attacked [191]. The differential attacks were further improved by Song et al. by increasing the number of rounds that can be attacked for ciphers with block sizes 96 and 128 bits, and reducing the attack complexities in the remaining cases [192]. In 2016, Feng et al. published fault attacks on SPECK ciphers which were better than the earlier attacks of the same type [193].

In 2017, Liu et al. proposed a SAT / SMT model for rotational-XOR cryptanalysis in ARX primitives and used it to present distinguishers on reduced-round SPECK with block sizes 32 and 48 bits which have better probabilities than the previously known differential characteristics [194]. In 2019, Ge et al. reported correlation power analysis on unprotected software implementations of SPECK ciphers [195]. In the same

year, Hou et al. presented a tool to perform automated differential fault analysis on the software implementations of block ciphers and used the same to attack the implementations of SPECK ciphers [196]. Reduced-round SPECK ciphers were subjected to a few more attacks during the years 2019 and 2020 such as impossible differential cryptanalysis, differential cryptanalysis, integral cryptanalysis and linear cryptanalysis [197, 198, 199, 200, 201, 202, 203].

A comparative analysis of the best-known key recovery attacks on the SPECK ciphers in chronological order, including the attacks presented in this Chapter, is listed in Table 4.1. The first parameter used to compare the attacks is the number of rounds of SPECK that can be targeted; greater the number of rounds, better is the attack. Among the attacks that target the same number of rounds, the best will be the one that requires the minimum number of fault injections, and have the minimum data and time complexities. It can be seen from Table 4.1 that the number of rounds targeted by the best-known non-implementation attacks ranges from 14 to 25 for different members of the SPECK family [192]. All the attacks on the full SPECK ciphers target their implementations, and the attacks presented in this Chapter are the best among them, as far as we know.

Processor flag cryptanalysis. In [129], Kelsey et al. introduced processor flag cryptanalysis, a new class of side-channel attacks. Nearly every modern microprocessor has the status register, a collection of flag bits that store information about the state of the processors and information on operations performed by their ALUs [130]. Carry flag is one such flag bit that indicates carry overflow in unsigned integer arithmetic. For instance, when two unsigned integers are added, the carry flag is set (to 1) if a carry is generated by the addition at the most significant bit position and the flag is cleared (i.e., 0) otherwise.

In 2000, the first known attack that exploits the carry flag was presented by Kelsey et al. on RC5 [129]. Later in 2002, Gomufflkiewicz et al. presented attacks on IDEA

Table 4.1: Comparative analysis of the best-known key recovery attacks (with a success rate of 99.99%) on the SPECK ciphers in chronological order

Block size (in bits)	Key size (in bits)	# rounds attacked / total	Attack requirements	Year
32	64	11 / 22	$2^{30.1}$ CPs, $2^{46.7}$ time	2014 [185]
		14 / 22	2^{31} CPs, 2^{63} time	2014 [187]
		22 / 22	6 faulty bits[1], 2^{48} time	2014 [188]
		22 / 22	5 faults[1], 2^{48} time	2015 [189]
		14 / 22	$2^{30.47}$ CPs, $2^{62.47}$ time	2016 [192]
		22 / 22	2^{14} data[2], 2^{59} time	2020 (This Chapter)
48	72	12 / 22	2^{45} CPs, $2^{45.3}$ time	2014 [185]
		12 / 22	2^{43} CPs, 2^{43} time	2014 [186]
		14 / 22	2^{41} CPs, 2^{65} time	2014 [187]
		22 / 22	8 faulty bits[1], 2^{48} time	2014 [188]
		22 / 22	5 faults[1], 2^{48} time	2015 [189]
		15 / 22	2^{46} CPs, 2^{70} time	2016 [191]
		15 / 22	$2^{45.31}$ CPs, $2^{69.31}$ time	2016 [192]
		22 / 22	2^{22} data[2], 2^{63} time	2020 (This Chapter)
48	96	12 / 23	2^{45} CPs, $2^{45.3}$ time	2014 [185]
		12 / 23	2^{43} CPs, 2^{43} time	2014 [186]
		15 / 23	2^{41} CPs, 2^{89} time	2014 [187]
		23 / 23	8 faulty bits[1], 2^{72} time	2014 [188]
		23 / 23	5 faults[1], 2^{72} time	2015 [189]
		16 / 23	2^{46} CPs, 2^{94} time	2016 [191]
		16 / 23	$2^{45.31}$ CPs, $2^{93.31}$ time	2016 [192]
		23 / 23	2^{22} data[2], 2^{87} time	2020 (This Chapter)
64	96	15 / 26	2^{61} CPs, $2^{61.1}$ time	2014 [185]
		16 / 26	2^{64} CPs, 2^{73} time	2014 [186]
		18 / 26	2^{61} CPs, 2^{93} time	2014 [187]
		26 / 26	11 faulty bits[1], 2^{64} time	2014 [188]
		26 / 26	6 faults[1], 2^{64} time	2015 [189]
		19 / 26	2^{63} CPs, 2^{95} time	2016 [191]
		19 / 26	$2^{61.56}$ CPs, $2^{93.56}$ time	2016 [192]
		26 / 26	2^{30} data[2], 2^{83} time	2020 (This Chapter)
64	128	15 / 27	2^{61} CPs, $2^{61.1}$ time	2014 [185]
		16 / 27	2^{64} CPs, 2^{73} time	2014 [186]
		19 / 27	2^{61} CPs, 2^{125} time	2014 [187]
		27 / 27	11 faulty bits[1], 2^{96} time	2014 [188]
		27 / 27	6 faults[1], 2^{96} time	2015 [189]

[1] Faults are induced in the last round.
[2] Number of encryptions are assumed to be well below the birthday bound.

Block size (in bits)	Key size (in bits)	# rounds attacked / total	Attack requirements	Year
64	128	20 / 27	2^{63} CPs, 2^{127} time	2016 [191]
		20 / 27	$2^{61.56}$ CPs, $2^{125.56}$ time	2016 [192]
		27 / 27	2^{30} data[2], 2^{115} time	2020 (This Chapter)
96	96	15 / 28	2^{89} CPs, $2^{89.1}$ time	2014 [185]
		16 / 28	2^{85} CPs, 2^{85} time	2014 [187]
		28 / 28	16 faulty bits[1], 2^{48} time	2014 [188]
		28 / 28	7 faults[1], 2^{48} time	2015 [189]
		19 / 28	2^{88} CPs, 2^{88} time	2016 [191]
		20 / 28	$2^{95.94}$ CPs, $2^{95.94}$ time	2016 [192]
		28 / 28	2^{46} data[2], 2^{75} time	2020 (This Chapter)
96	144	16 / 29	$2^{90.9}$ CPs, $2^{135.9}$ time	2014 [185]
		17 / 29	2^{85} CPs, 2^{133} time	2014 [187]
		29 / 29	16 faulty bits[1], 2^{96} time	2014 [188]
		29 / 29	7 faults[1], 2^{96} time	2015 [189]
		20 / 29	2^{88} CPs, 2^{136} time	2016 [191]
		21 / 29	$2^{95.94}$ CPs, $2^{143.94}$ time	2016 [192]
		29 / 29	2^{46} data[2], 2^{123} time	2020 (This Chapter)
128	128	16 / 32	2^{116} CPs, $2^{111.1}$ time	2014 [185]
		17 / 32	2^{113} CPs, 2^{113} time	2014 [187]
		32 / 32	22 faulty bits[1], 2^{64} time	2014 [188]
		32 / 32	8 faults[1], 2^{64} time	2015 [189]
		22 / 32	2^{120} CPs, 2^{120} time	2016 [191]
		23 / 32	$2^{125.35}$ CPs, $2^{125.35}$ time	2016 [192]
		32 / 32	2^{62} data[2], 2^{99} time	2020 (This Chapter)
128	192	18 / 33	$2^{125.9}$ CPs, $2^{182.7}$ time	2014 [185]
		18 / 33	2^{113} CPs, 2^{177} time	2014 [187]
		33 / 33	22 faulty bits[1], 2^{128} time	2014 [188]
		33 / 33	8 faults[1], 2^{128} time	2015 [189]
		23 / 33	2^{120} CPs, 2^{184} time	2016 [191]
		24 / 33	$2^{125.35}$ CPs, $2^{189.35}$ time	2016 [192]
		33 / 33	2^{62} data[2], 2^{163} time	2020 (This Chapter)
128	256	18 / 34	$2^{125.9}$ CPs, $2^{182.7}$ time	2014 [185]
		19 / 34	2^{113} CPs, 2^{241} time	2014 [187]
		34 / 34	22 faulty bits[1], 2^{192} time	2014 [188]
		34 / 34	8 faults[1], 2^{192} time	2015 [189]
		24 / 34	2^{120} CPs, 2^{248} time	2016 [191]
		25 / 34	$2^{125.35}$ CPs, $2^{253.35}$ time	2016 [192]
		34 / 34	2^{62} data[2], 2^{227} time	2020 (This Chapter)

and Twofish [134]. These attacks exploited the Hamming weight of the sequence of carry bits and the authors conjectured that simple power analysis can be used to extract this information. Subsequently, in 2008, Fouque et al. presented carry flag attacks on public key implementations of RSA and ECC based cryptosystems with exponent randomisation countermeasure [133]. They demonstrated that the carry flag can be detected by studying the electromagnetic side-channel of the implementation using differential power analysis. In 2009, Fouque et al. presented another carry flag attack which could recover the secret keys used in public key schemes such as DSA and ECDSA signature schemes, and Schnorr and GPS authentication and signature schemes [132]. In order to detect the carry, they injected a fault during the computation of the carry using methods like glitch injection, clock speed up or laser fault injection. The most recent carry flag attack, which is on Streebog, was presented by Sekar in 2015 [131].

Contributions of this work. SPECK is a cipher of importance as it is an ISO standard for RFID devices and a potential candidate to secure lightweight applications. This gave us enough motivation to examine its security. After unsuccesfully trying several cryptanalytic methods to analyse SPECK, it was eventually found that SPECK is vulnerable to carry flag attacks. This technique, originally proposed by Kelsey et al. [129], is fairly well-known and thoroughly explored in a few papers [131, 132, 133, 134]. Hence, we were motivated to further examine the vulnerability of SPECK to this class of attacks. Despite the use of modular addition in the round function of SPECK, to the best of our knowledge, there are no published results analysing the resistance of unprotected SPECK implementations to carry flag attacks. Our attacks, that work on the full ciphers, at the best recover the key in 2^{59} time with 2^{14} data corresponding to a success rate of 99.99%. The only other full-round attacks on SPECK are due to [188], [189] and [193], however, these are based on much stronger assumptions. Our attack model uses a weak assumption that the attacker can either detect the carry flag

at the end of encryption or key expansion, or detect the bit-flips in the carry flag after the modular addition in the final round. This makes our attacks more feasible when compared to the fault attacks of [188, 189, 193]. The complexities of our key recovery attacks for a success rate of 99.99% are given in Table 4.1.

Organisation of the Chapter. The remaining Chapter is organised as follows. Section 4.2 describes the ciphers. The motivational observations for SPECK are presented in Sect. 4.3. Our attacks on the encryption phase and key expansion phase of SPECK are presented in Sect. 4.4 and Sect. 4.5, respectively. The methods to detect the carry flag, scope of improving the attack complexities and countermeasures are discussed in Sect. 4.6. We conclude in Sect. 4.7.

4.2 Specifications of the Ciphers

4.2.1 SPECK$_{m,n}$

The lightweight block cipher SPECK$_{m,n}$ uses an n-bit key K, and has a block size of m bits and word size of $\frac{m}{2}$ bits. The SPECK$_{m,n}$ encryption consists of r rounds and the $(i+1)$th round function is given by $E_{k_i}(a_i, b_i) = \big(((a_i \ggg \alpha) \boxplus_{(\frac{m}{2})} b_i) \oplus k_i,$ $(((a_i \ggg \alpha) \boxplus_{(\frac{m}{2})} b_i) \oplus k_i) \oplus (b_i \lll \beta)\big)$, where k_i is the$(i+1)$th round key of size $\frac{m}{2}$ bits, a_i and b_i are the $\frac{m}{2}$-bit inputs of the $(i+1)$th round, and α and β are round constants. The inverse of the $(i+1)$th round function, which will be used for decryption, is given by $D_{k_i}(a_{i+1}, b_{i+1}) = \big(((a_{i+1} \oplus k_i) \boxminus_{(\frac{m}{2})} (a_{i+1} \oplus b_{i+1}) \ggg \beta) \lll \alpha, (a_{i+1} \oplus b_{i+1}) \ggg \beta\big)$, where a_{i+1} and b_{i+1} are the $\frac{m}{2}$-bit outputs of the $(i+1)$th round. The $(i+1)$th round function of SPECK$_{m,n}$ is shown in Figure 4.1. There are 10 choices for the pair (m, n) and the parameters r, α and β corresponding to each of them are given in Table 4.2. SPECK$_{m,n}$ operates in 3 phases: key expansion, encryption and decryption.

Key expansion: The key schedule algorithm takes K as input and generates the subkeys $k_0, k_1, \ldots, k_{r-1}$.

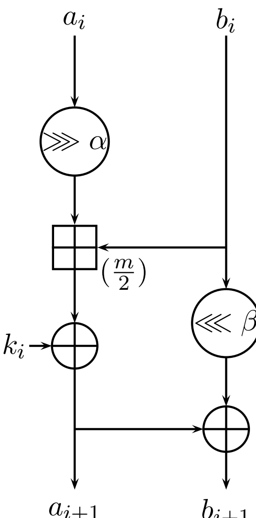


Figure 4.1: The $(i+1)$th round of SPECK$_{m,n}$

Table 4.2: Parameters of SPECK$_{m,n}$

(m,n)	r	α	β
$(32, 64)$	22	7	2
(48, 72)	22	8	3
(48, 96)	23	8	3
(64, 96)	26	8	3
(64, 128)	27	8	3
(96, 96)	28	8	3
(96, 144)	29	8	3
(128, 128)	32	8	3
(128, 192)	33	8	3
(128, 256)	34	8	3

Algorithm 8 Key schedule of SPECK$_{m,n}$

Require: K

Ensure: The subkeys $k_0, k_1, \ldots, k_{r-1}$

1: $k_0 = K_{(\frac{m}{2}-1)} \parallel K_{(\frac{m}{2}-2)} \parallel \ldots \parallel K_{(0)}$;
2: $w = \frac{2n}{m}$;
3: **for** i = 0 to $w-2$ **do**
4: $\quad l_i = K_{\left(\frac{(i+2)m}{2}-1\right)} \Big\| K_{\left(\frac{(i+2)m}{2}-2\right)} \Big\| \ldots \Big\| K_{\left(\frac{(i+1)m}{2}\right)}$;
5: **endfor**
6: **for** i = 0 to $r-2$ **do**
7: $\quad (l_{i+w-1},\, k_{i+1}) \leftarrow E_i(l_i, k_i)$;
8: **endfor**

Encryption: The encryption algorithm takes as inputs $k_0, k_1, \ldots, k_{r-1}$ and the plaintext block $P_1 \parallel P_2$ and generates the ciphertext block $C_1 \parallel C_2$ — P_1, P_2, C_1 and C_2 are $\frac{m}{2}$-bit words.

Algorithm 9 Encryption of SPECK$_{m,n}$

Require: The subkeys $k_0, k_1, \ldots, k_{r-1}$ and the plaintext block $P_1 \parallel P_2$

Ensure: The ciphertext block $C_1 \parallel C_2$

1: $u = P_1$;
2: $v = P_2$;
3: **for** i = 0 to $r-1$ **do**
4: $\quad (C_1,\, C_2) \leftarrow E_{k_i}(u, v)$;
5: $\quad u \leftarrow C_1$;
6: $\quad v \leftarrow C_2$;
7: **endfor**

Decryption: The decryption algorithm takes as inputs $k_0, k_1, \ldots, k_{r-1}$ and the ciphertext block $C_1 \parallel C_2$ and generates the plaintext block $P_1 \parallel P_2$.

Algorithm 10 Decryption of SPECK$_{m,n}$

Require: The subkeys $k_0, k_1, \ldots, k_{r-1}$ and the ciphertext block $C_1 \parallel C_2$
Ensure: The plaintext block $P_1 \parallel P_2$

1: $u = C_1$;
2: $v = C_2$;
3: **for** $i = r - 1$ to 0 **do**
4: $(P_1, P_2) \leftarrow D_{k_i}(u, v)$;
5: $u \leftarrow P_1$;
6: $v \leftarrow P_2$;
7: **endfor**

4.3 Motivational Observation

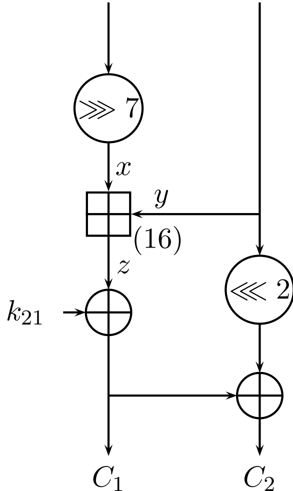


Figure 4.2: Final round of encryption of SPECK$_{32,64}$

The final round of encryption of SPECK$_{32,64}$ is shown in Figure 4.2. Let x and y, which are distributed uniformly at random, be the inputs to the modular addition in the final round of SPECK$_{32,64}$, z be the sum of x and y modulo 2^{16}, c_{i+1} be the carry generated at the $(i + 1)$th bit position of the addition modulo operation, where $0 \leq i \leq 15$ (for instance, outgoing carry at the LSB position is denoted by c_1), c_{out} be the carry flag at

the end of the encryption and $C_1 \parallel C_2$ be the ciphertext block generated. We assume that c_{out} equals c_{16}. The allowed values of the Boolean variables $x_{(i)}$, $y_{(i)}$, c_i, $z_{(i)}$ and c_{i+1} are tabulated in Table 4.3. The truth table clearly shows that c_i and $z_{(i)} \oplus 1$ are biased towards c_{i+1}.

Based on this observation, we compute the bias of $z_{(i)} \oplus 1$ towards c_{out} for any $i \in \{15, 14, \ldots, 0\}$. It is reasonable to assume that $x_{(i)}$, $y_{(i)}$ and c_i are independent, and all the possible outcomes given in Table 4.3 are mutually exclusive. Let $\Pr(x_{(i)} = 0) = \Pr(y_{(i)} = 0) = \frac{1}{2}$ and $\Pr(c_i = 0) = \rho_i$. From Table 4.3, we deduce the following:

$$\Pr(c_i \oplus c_{i+1} = 0) = 3 \cdot \frac{\rho_i}{4} + 3 \cdot \frac{1 - \rho_i}{4} = \frac{3}{4}, \tag{4.1}$$

$$\Pr(z_{(i)} \oplus c_{i+1} = 1) = 3 \cdot \frac{\rho_i}{4} + 3 \cdot \frac{1 - \rho_i}{4} = \frac{3}{4}. \tag{4.2}$$

Let $\Pr(c_{i+1} \oplus c_{out} = 0) = q_{i+1}$. From (4.1) and (4.2), we get:

$$\begin{aligned}
\Pr(c_i \oplus c_{out} = 0) &= \Pr(c_i \oplus c_{i+1} = 0) \cdot \Pr(c_{i+1} \oplus c_{out} = 0) \\
&\quad + \Pr(c_i \oplus c_{i+1} = 1) \cdot \Pr(c_{i+1} \oplus c_{out} = 1) \\
&= \frac{3}{4} q_{i+1} + \frac{1}{4}(1 - q_{i+1}) \\
&= \frac{2q_{i+1} + 1}{4},
\end{aligned} \tag{4.3}$$

$$\begin{aligned}
\Pr(z_{(i)} \oplus c_{out} = 1) &= \Pr(z_{(i)} \oplus c_{i+1} = 1) \cdot \Pr(c_{i+1} \oplus c_{out} = 0) \\
&\quad + \Pr(z_{(i)} \oplus c_{i+1} = 0) \cdot \Pr(c_{i+1} \oplus c_{out} = 1) \\
&= \frac{3}{4} q_{i+1} + \frac{1}{4}(1 - q_{i+1}) \\
&= \frac{2q_{i+1} + 1}{4}.
\end{aligned} \tag{4.4}$$

Since $c_{16} = c_{out}$, we get:

$$\Pr(z_{(15)} \oplus c_{out} = 1) = \frac{3}{4}, \tag{4.5}$$

$$\Pr(c_{15} \oplus c_{out} = 0) = \frac{3}{4}. \tag{4.6}$$

Equations (4.3), (4.4) and (4.6) gives:

$$\Pr(z_{(14)} \oplus c_{out} = 1) = \frac{5}{8}, \tag{4.7}$$

$$\Pr(c_{14} \oplus c_{out} = 0) = \frac{5}{8}. \tag{4.8}$$

Similarly, probabilities that $z_{(i)} \oplus c_{out} = 1$ for the remaining values of i can be computed by recursively applying (4.3) and (4.4), and are tabulated in Table 4.4.

Table 4.3: Truth table showing the allowed values of the Boolean variables $x_{(i)}, y_{(i)}$, c_i, $z_{(i)}$ and c_{i+1}

$x_{(i)}$	$y_{(i)}$	c_i	$z_{(i)}$	c_{i+1}
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

4.4 Key Recovery

In this Section, we will see how an attacker, who has access to the encryption device of SPECK$_{32,64}$, exploits the bias present in the distribution of $z_{(i)} \oplus c_{out}$ to recover K with

Table 4.4: Probabilities that $z_{(i)} \oplus c_{out} = 1$, for $15 \geq i \geq 0$, in SPECK$_{32,64}$

i	Pr($z_{(i)} \oplus c_{out} = 1$)
15	$0.5 + 2^{-2}$
14	$0.5 + 2^{-3}$
13	$0.5 + 2^{-4}$
12	$0.5 + 2^{-5}$
11	$0.5 + 2^{-6}$
10	$0.5 + 2^{-7}$
9	$0.5 + 2^{-8}$
8	$0.5 + 2^{-9}$
7	$0.5 + 2^{-10}$
6	$0.5 + 2^{-11}$
5	$0.5 + 2^{-12}$
4	$0.5 + 2^{-13}$
3	$0.5 + 2^{-14}$
2	$0.5 + 2^{-15}$
1	$0.5 + 2^{-16}$
0	$0.5 + 2^{-17}$

a complexity lesser than that of exhaustive search. At the end of each encryption, the attacker observes the ciphertext as well as the carry flag. Then she makes a guess $\widehat{k}_{21(i)}$ for $k_{21(i)}$, where $15 \geq i \geq 0$. Based on the guess, she computes $\widehat{z}_{(i)} := C_{1(i)} \oplus \widehat{k}_{21(i)}$ and in turn $b_i := \widehat{z}_{(i)} \oplus c_{out}$. Since $z_{(i)} \oplus c_{out}$ is biased towards 1, her guess will be right only when b_i is also biased towards 1.

Let N denote the number of b_i's available to the attacker (collected from different cipher texts), β_i denote the distribution of $B_i := \sum_N b_i$, $p_i = \Pr(b_i = 1)$, β'_i denote the distribution of B_i if b_i's were biased towards 0 and $p'_i = 1 - p_i$. For any given i, if the b_i's are independent and identically distributed random variables (i.i.d.) then B_i has a binomial distribution. The means (μ_i, μ'_i) and standard deviations (σ_i, σ'_i) of the distributions β_i, β'_i are given by: $\mu_i = Np_i$, $\mu'_i = Np'_i$, $\sigma_i = \sqrt{Np_i(1-p_i)}$ and $\sigma'_i = \sqrt{Np'_i(1-p'_i)}$. If N is large, one can approximate each binomial distribution with the normal distribution having the same mean and standard deviation.

Given this, if $|\mu_i - \mu'_i| > t_i(\sigma_i + \sigma'_i) \implies N > 0.25{t_i}^2/(p_i - 0.5)^2$, the attacker can determine whether b_i is biased towards 1 or 0 with success probability ω_i, where ω_i is

the value given by the cumulative distribution function at $\mu_i + t_i\sigma_i$.[3] If t_i is 3.28, the bias can be determined with 99.99% success rate (since the cumulative distribution function gives the value 0.9999 at $\mu_i + 3.62\sigma_i$) and 0.01% false positive rate.[4] The number of samples required to determine if b_i is biased to 1 or 0, for any i such that $15 \geq i \geq 0$, with an upper bound 2^{32} (since the block size is 32) are listed in Table 4.5. The attacker will be able to recover the 13 most significant bits of k_{21} with more than 99.99% success rate by observing $2^{29.7}$ encryptions. The values of $p_i - 0.5$, t_i and the success probabilities ω_i, for any i such that $15 \geq i \geq 0$, when $N = 2^{30}$ are listed in Table 4.6.

If k_{21}, k_{20}, k_{19} and k_{18} are known, the key schedule of SPECK$_{32,64}$ can be reversed using Algorithm 11. Therefore, the unrecovered bits of k_{21} along with k_{20}, k_{19} and k_{18} can be obtained through brute force with a complexity of 2^{51} time which in turn leads to the recovery of K.

Table 4.5: Data requirements for SPECK$_{32,64}$ (corresponding to 0.9999 success probability) to determine if b_i is biased to 1 or 0, for $15 \geq i \geq 0$

i	$\log_2(\#samples)$
15	5.7
14	7.7
13	9.7
12	11.7
11	13.7
10	15.7
9	17.7
8	19.7
7	21.7
6	23.7
5	25.7
4	27.7
3	29.7
2	31.7
1	32
0	32

[3]The cumulative distribution function $\phi(x)$ of the standard normal distribution is given by $\phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} e^{-t^2/2} dt$.

[4]Since $\phi(\infty) = 1$, 100% success rate will be theoretically possible only at ∞.

Table 4.6: Values of $p_i - 0.5$, t_i and the success probabilities ω_i corresponding to $N = 2^{30}$, for $15 \geq i \geq 0$

i	$p_i - 0.5$	t_i	ω_i
15	2^{-2}	16384	~ 1
14	2^{-3}	8192	~ 1
13	2^{-4}	4096	~ 1
12	2^{-5}	2048	~ 1
11	2^{-6}	1024	~ 1
10	2^{-7}	512	~ 1
9	2^{-8}	256	~ 1
8	2^{-9}	128	~ 1
7	2^{-10}	64	~ 1
6	2^{-11}	32	~ 1
5	2^{-12}	16	~ 1
4	2^{-13}	8	~ 1
3	2^{-14}	4	0.9999
2	2^{-15}	2	0.9772
1	2^{-16}	1	0.8413
0	2^{-17}	0.5	0.6915

Algorithm 11 Key recovery of SPECK$_{m,n}$

Require: The subkeys $k_j, k_{j+1}, \ldots, k_{j+\frac{2n}{m}-1}$ where $0 \leq j \leq r - \frac{2n}{m}$
Ensure: K

1: $w = \frac{2n}{m}$;
2: **for** $i = w - 2$ to 0 **do**
3: $l_{j+i+w-1} \leftarrow k_{j+i+1} \oplus (k_{j+i} \lll 2)$;
4: **endfor**
5: **for** $i = j + w - 2$ to 0 **do**
6: $(l_i, k_i) \leftarrow D_i(l_{i+w-1}, k_{i+1})$;
7: **endfor**
8: $K \leftarrow l_{w-2} \parallel l_{w-3} \parallel \ldots \parallel l_0 \parallel k_0$;

4.4.1 Experimental Verification

In order to verify our analysis, the attack on SPECK$_{32,64}$ was simulated. We encrypted 2^{23} plaintext blocks generated uniformly at random using an arbitrary key and observed

Algorithm 12 Subkey recovery of SPECK$_{32,64}$

Require: The ciphertext word C_1 and the output carry c_{out} generated from 2^{23} encryptions

Ensure: The subkey k_{21}

1: **for** i = 15 to 0 **do**
2: $Z_i = 0$;
3: $k_{21(i)} \leftarrow 0$;
4: **endfor**
5: $n = 2^{23}$;
6: **do** until n ciphertexts are generated
7: **for** i = 15 to 0 **do**
8: $b_i = C_{1(i)} \oplus k_{21(i)} \oplus c_{out}$;
9: $B_i = B_i + b_i$;
10: **endfor**
11: **enddo**
12: **for** i = 15 to 0 **do**
13: $\epsilon_i = 2^{i-17}$; [a]
14: $t_i = 2\epsilon_i\sqrt{n}$;
15: $\sigma_i = \sqrt{n(0.25 - \epsilon_i^2)}$;
16: $\alpha_i = n(0.5 + \epsilon_i) - t_i\sigma_i + 1$; [b]
17: **if** $B_i < \alpha_i$
18: $k_{21(i)} \leftarrow 1$;
19: **endif**
20: **endfor**

[a] $\epsilon_i = |p_i - 0.5|$

[b] α_i is the critical value such that the confidence intervals of the distributions β_i and β_i' do not overlap. At $\alpha_i - 1$, they overlap.

the output carry (c_{out}) generated at the end of each encryption.[5] Using Algorithm 12, we set $k_{21(i)}$ as 0, where $15 \geq i \geq 0$, if B_i was lying within the confidence interval of β_i, or 1 otherwise.[6] This process was repeated 2×10^4 times. The success rates at which we were able to recover each bit of the subkey are tabulated in Table 4.7. The C

[5] The 64 least significant bits of $\xi_\kappa(C)$ form an arbitrary key (for e.g., `0x23f7290ff115eda1`) where ξ_κ is AES encryption function with a random key κ and C is some constant.

[6] The critical value used to distinguish between β_i and β_i' has to ensure that the confidence intervals of the distributions do not overlap.

program which can be used to recover the 10 most significant bits of the final subkey of SPECK$_{32,64}$ is given in Figure 4.3.

Table 4.7: Success rates of our experiments on SPECK$_{32,64}$ to recover $k_{21(i)}$ with 2^{23} samples, for $15 \geq i \geq 0$

i	Success rate (%)
15	100
14	100
13	100
12	100
11	100
10	100
9	100
8	100
7	100
6	99.84
5	92.28
4	75.91
3	62.98
2	57.17
1	54.09
0	49.72

4.4.2 Reducing the Time Complexity of the Attack

The time complexity of our attack can be reduced by utilising the information about the bias in the distribution of $z_{(i)} \oplus c_{out}$ while performing exhaustive search of the unrecovered bits of k_{21}. For instance, if 2^{30} b_i's are available to the attacker then the 13 most significant bits of k_{21} can be recovered with 99.99% success rate where as the recovery of $k_{21(2)}$, $k_{21(1)}$ and $k_{21(0)}$ will only have 97.72%, 84.13% and 69.15% success rates, respectively (see Table 4.6). In other words, at least one of the three least significant bits of k_{21} can be recovered with a success rate of 99.89% or at least two of them can be recovered with 94.26% success rate (see (4.9) and (4.10)). If we assume that one of the three least significant bits of k_{21} computed using Algorithm 12 is correct, only 7 out of the 8 possible choices of the three bits need to be checked while doing brute

Figure 4.3: Program to recover the 10 most significant bits of the final subkey of $\text{SPECK}_{32,64}$

```
#include "fetch.h"
int main(void)
{
    uint32_t n = (1<<23); // Number of samples available
    uint16_t C1[n], cout[n], subkey21;

    /* Function to fetch the most significant 16 bits of the ciphertexts from n encryptions
    and the corresponding carry flags */
    fetch(C1, cout, n);

    double p, q, r, t;
    int crit[16], i, j, count[16];
    for (j=15;j>-1;j--)
    {
        p = 0.5 + pow(2,-(2+15-j));
        q = 1 - p;
        t = 2*(p-0.5)*sqrt((double)n);
        r = (t * sqrt((double)n * p * q)) - 1;
        crit[j] = (int)(n*p) - (int)ceil(r);
        count[j] = 0;
    }
    for (i=0;i<n;i++)
    {
        for (j=15;j>-1;j--)
        {
            count[j] += (((C1[i] >> j) & 0x1) ^ cout[i]);
        }
    }
    subkey21 = 0;
    for (j=15;j>5;j--)
    {
        if (count[j] < crit[j]) subkey21 ^= (1<<j);
    }
    Printf("Most significant 10 bits of the final subkey have been recovered.\n");
    printf("Recovered bits of the subkey in hexadecimal format: %03x \n", subkey21 >> 6);
    return 0;
}
```

force. Similarly, exhaustive search can be limited to 4 out of the 8 possible choices, if it is assumed that any two out of the three least significant bits have been correctly computed. Therefore, the time complexity of the full key recovery attack can be reduced to a factor of $\frac{7}{8}$ with 99.89% success rate or to a factor of $\frac{1}{2}$ with 94.26% success rate. In order to prevent collision attacks [204], designers recommend that "the number of blocks encrypted using a single key for an n-bit block cipher should be kept well under $2^{\frac{n}{2}}$" [205]. Taking this into consideration, it is assumed that 2^{14} b_i's are only available to the attacker which, in turn, will increase the time complexity of our attack.[7] The remaining versions of SPECK can also be attacked similarly and the complexities of our attacks on $\text{SPECK}_{m,n}$ for the success rates 99.99%, 99.89% and 94.26% are listed in Table 4.8.

[7] $\text{SPECK}_{m,n}$ is assumed to encrypt at the most $2^{\frac{m}{2}-2}$ blocks using a single key so that the collision probability will be below 2^{-5}.

Table 4.8: Complexities of the full key recovery attacks on $\text{SPECK}_{m,n}$ corresponding to the success rates (SR) of 99.99%, 99.89% and 94.26%

	SR = 99.99%		SR = 99.89%		SR = 94.26%	
Cipher	Data	Time	Data	Time	Data	Time
$\text{SPECK}_{32,64}$	2^{14}	2^{59}	2^{14}	$2^{58.81}$	2^{14}	2^{58}
$\text{SPECK}_{48,72}$	2^{22}	2^{63}	2^{22}	$2^{62.81}$	2^{22}	2^{62}
$\text{SPECK}_{48,96}$	2^{22}	2^{87}	2^{22}	$2^{86.81}$	2^{22}	2^{86}
$\text{SPECK}_{64,96}$	2^{30}	2^{83}	2^{30}	$2^{82.81}$	2^{30}	2^{82}
$\text{SPECK}_{64,128}$	2^{30}	2^{115}	2^{30}	$2^{114.81}$	2^{30}	2^{114}
$\text{SPECK}_{96,96}$	2^{46}	2^{75}	2^{46}	$2^{74.81}$	2^{46}	2^{74}
$\text{SPECK}_{96,144}$	2^{46}	2^{123}	2^{46}	$2^{122.81}$	2^{46}	2^{122}
$\text{SPECK}_{128,128}$	2^{62}	2^{99}	2^{62}	$2^{98.81}$	2^{62}	2^{98}
$\text{SPECK}_{128,192}$	2^{62}	2^{163}	2^{62}	$2^{162.81}$	2^{62}	2^{162}
$\text{SPECK}_{128,256}$	2^{62}	2^{227}	2^{62}	$2^{226.81}$	2^{62}	2^{226}

$$\omega_a = 1 - ((1-\omega_2)(1-\omega_1)(1-\omega_0)) = 0.9989\,, \tag{4.9}$$

$$\begin{aligned}\omega_b = {} & 1 - ((1-\omega_2)(1-\omega_1)(1-\omega_0)) - (\omega_2(1-\omega_1)(1-\omega_0)) \\ & -((1-\omega_2)\omega_1(1-\omega_0)) - ((1-\omega_2)(1-\omega_1)\omega_0) = 0.9426\,,\end{aligned} \tag{4.10}$$

where ω_2, ω_1 and ω_0 are the success probabilities to recover $k_{21(2)}$, $k_{21(1)}$ and $k_{21(0)}$, respectively, and ω_a and ω_b are the success probabilities to recover at least one of them and at least two of them, respectively.

4.5 Attack on Key Schedule

In this Section, we will see how the carry generated at the end of the key expansion phase can be exploited to reduce the complexity of our attack on $\text{SPECK}_{32,64}$. Since the round functions used in key schedule and encryption algorithms are the same, weakness similar to the one mentioned in Sect. 4.3 will be present in the key expansion phase too. The final round of the key expansion phase of $\text{SPECK}_{32,64}$ is shown in Figure 4.4. Let x

and y, which are distributed uniformly at random, be the inputs to the modular addition in the final round of the key schedule, z be the sum of x and y modulo 2^{16}, c_{i+1} be the carry generated at the $(i + 1)$th bit position of the addition modulo operation, where $0 \leq i \leq 15$ (for example, outgoing carry at the LSB position is denoted by c_1), and d_{out} be the carry flag at the end of the key schedule. The allowed values of the Boolean variables $x_{(i)}$, $y_{(i)}$, c_i, $z_{(i)}$ and c_{i+1} are same as those tabulated in Table 4.3 and from it, we find that $\Pr(y_{(i)} = c_{i+1}) = \frac{3}{4}$. We assume that d_{out} equals c_{16} and therefore we get, $\Pr(y_{(15)} = d_{out}) = \frac{3}{4}$. In other words, if the attacker is able to observe the carry flag at the end of key expansion phase, the most significant bit of $y = k_{20}$ can be recovered immediately with a success rate of 75%. Combining this information with the attack of Sect. 4.4.2, we deduce that from the set of bits including $k_{20(15)}$ and the three least significant unknown bits of k_{21}, any one can be recovered with a success rate of 99.97%, any two bits can be recovered with 98.48% success rate or any three bits can be recovered with 84.91% success rate. Similar attacks can be built on the remaining versions of SPECK too. The complexities of the key recovery attacks on SPECK$_{m,n}$ corresponding to the success rates of 99.97%, 98.48% and 84.91% are listed in Table 4.9.[8]

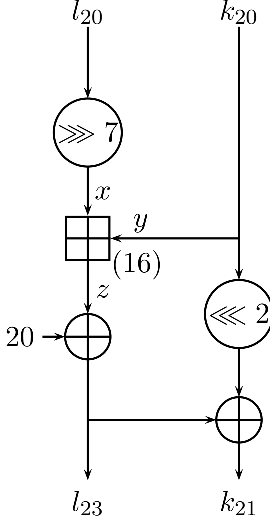


Figure 4.4: Final round of the key expansion phase of SPECK$_{32,64}$

[8]Data requirements are same as those listed in Table 4.8.

Table 4.9: Complexities of the full key recovery attacks on $\text{SPECK}_{m,n}$ corresponding to the success rates (SR) of 99.97%, 98.48% and 84.91%, under the assumption that the carry flag at the end of the key expansion phase is known

	SR = 99.97%		SR = 98.48%		SR = 84.91%	
Cipher	Data	Time	Data	Time	Data	Time
$\text{SPECK}_{32,64}$	2^{14}	$2^{58.9}$	2^{14}	$2^{58.46}$	2^{14}	$2^{57.32}$
$\text{SPECK}_{48,72}$	2^{22}	$2^{62.9}$	2^{22}	$2^{62.46}$	2^{22}	$2^{61.32}$
$\text{SPECK}_{48,96}$	2^{22}	$2^{86.9}$	2^{22}	$2^{86.46}$	2^{22}	$2^{85.32}$
$\text{SPECK}_{64,96}$	2^{30}	$2^{82.9}$	2^{30}	$2^{82.46}$	2^{30}	$2^{81.32}$
$\text{SPECK}_{64,128}$	2^{30}	$2^{114.9}$	2^{30}	$2^{114.46}$	2^{30}	$2^{113.32}$
$\text{SPECK}_{96,96}$	2^{46}	$2^{74.9}$	2^{46}	$2^{74.46}$	2^{46}	$2^{73.32}$
$\text{SPECK}_{96,144}$	2^{46}	$2^{122.9}$	2^{46}	$2^{122.46}$	2^{46}	$2^{121.32}$
$\text{SPECK}_{128,128}$	2^{62}	$2^{98.9}$	2^{62}	$2^{98.46}$	2^{62}	$2^{97.32}$
$\text{SPECK}_{128,192}$	2^{62}	$2^{162.9}$	2^{62}	$2^{162.46}$	2^{62}	$2^{161.32}$
$\text{SPECK}_{128,256}$	2^{62}	$2^{226.9}$	2^{62}	$2^{226.46}$	2^{62}	$2^{225.32}$

4.6 Discussion

Vulnerable implementations. The attacks presented in this Chapter are based on one of the following adversarial assumptions:

- **AA1:** The outgoing carries at the MSB position of modular additions in the final round of encryption phase and key expansion phase agree with the carry flags at the end of the respective phases, and the attacker can detect them.
- **AA2:** The attacker can detect the bit-flips in the carry flag after the modular addition of the final round of encryption phase.

In [206], Beaulieu et al. presented a one-bit-per-clock implementation of SPECK on ASIC in which the modular addition is implemented using a full adder circuit with a carry flag. From this reference implementation, it is inferred that the carry flag is not altered after the modular addition making it vulnerable to our attack as per AA1.

Based on the reference pseudocode provided in [207], SPECK can have two types of software implementations which we call Implementation A and Implementation B. The pseudocodes of the final rounds of the two software implementations of SPECK are given in Table 4.10. The left circular shift operation that follows the modular addition in

Implementation A alters the carry flag making it invulnerable going by AA1. A possible way to keep the carry flag unaffected after the targeted modular addition is to skip the processor instructions related to the left circular shift operation of the final round by introducing timing or voltage glitches as presented in [208, 209]. Since the skipped instructions do not affect C_1, our attack still works.

Table 4.10: Pseudocodes of the final rounds of the two software implementations of SPECK, where r is the number of rounds and (x, y) are the inputs to the final round

Implementation A	Implementation B
1: $x \leftarrow (x \ggg \alpha)$	1: $x \leftarrow (x \ggg \alpha)$
2: $x \leftarrow x + y$	2: $tmp \leftarrow (y \lll \beta)$
3: $x \leftarrow x \oplus k_{r-1}$	3: $x \leftarrow x + y$
4: $y \leftarrow (y \lll \beta)$	4: $x \leftarrow x \oplus k_{r-1}$
5: $y \leftarrow y \oplus x$	5: $y \leftarrow tmp \oplus x$
6: $(C_1, C_2) \leftarrow (x, y)$	6: $(C_1, C_2) \leftarrow (x, y)$

To validate it, we compiled the reference implementation of SPECK for 8-bit AVR microcontrollers [210], which is a secure implementation as per AA1, available at the University of Luxembourg Fair Evaluation of Lightweight Cryptographic Systems (FELICS) project [211] after manually removing the processor instructions related to the left circular shift operation of the final round. The execution of the generated hex file in Atmega128 microcontroller was simulated using the AVR simulator *simavr* and we could confirm that the outgoing carry at the MSB position of modular addition in the final round agrees with the carry flag at the end of encryption as shown in Figure 4.5. Since both the implementations provided in Table 4.10 generate the same output, malicious updating of the microcontroller firmware to include the vulnerable code might remain undetected leading to another possible way to attack an otherwise secure implementation.

The attacker who observes the ciphertext block $C_1 \parallel C_2$ can compute the input to the left circular shift operation as $(C_1 \oplus C_2) \ggg \beta$. If the input is known, she can compute the carry flag at the end of left circular shift. If AA2 is valid, the attacker detects the

Figure 4.5: Simulation of the SPECK implementation for AVR microcontrollers available in the FELICS project [211] to confirm that the outgoing carry at the MSB position of modular addition in the final round agrees with the carry flag at the end of encryption, if the left circular shift operation of the final round is skipped

```
#define SPECK_ENC_ROUND2(x3, x2, x1, x0, y3, y2, y1, y0, k3, k2, k1, k0) \
   LOAD_KEY(k3, k2, k1, k0)                                              \
   ADD(x3, x2, x1, x0, y3, y2, y1, y0)                                   \
   XOR_SHIFT(k3, k2, k1, k0, x3, x2, x1, x0)                             \
   LCS3(y3, y2, y1, y0)                                                  \
   XOR(y3, y2, y1, y0, k3, k2, k1, k0)                                   \
   LOAD_KEY(x3, x2, x1, x0)                                              \
   ADD(k3, k2, k1, k0, y3, y2, y1, y0)                                   \
   XOR_SHIFT(x3, x2, x1, x0, k3, k2, k1, k0)                             \
   LCS3(y3, y2, y1, y0)   <-- Operation to be skipped                    \
   XOR(y3, y2, y1, y0, x3, x2, x1, x0)
```

```
/* Left circular shift in place */
#define LCS1_(x3, x2, x1, x0)
  lsl x0
  rol x1
  rol x2
  rol x3
  adc x0, r1

#define LCS3_(x3, x2, x1, x0)
  LCS1_(x3, x2, x1, x0)
  LCS1_(x3, x2, x1, x0)
  LCS1_(x3, x2, x1, x0)

#define LCS3(x3, x2, x1, x0)
  STR(LCS3_(x3, x2, x1, x0))
```

```
int main()
{
        stdout = &mystdout;
        uint8_t key[KEY_SIZE] =
        {
        /* key = 0x03020100, 0x0b0a0908, 0x13121110 */
         0x00, 0x01, 0x02, 0x03,
         0x08, 0x09, 0x0a, 0x0b,
        0x10, 0x11, 0x12, 0x13
        };
        uint8_t pt[BLOCK_SIZE] = {0x65,0x71,0x58,0x79,0x90,0xfd,0x25,0x7a};
        uint8_t state[BLOCK_SIZE], roundKeys[ROUND_KEYS_SIZE], i, carry;

        InitializeState(pt,state);
        RunEncryptionKeySchedule(key, roundKeys);
        Encrypt(state, roundKeys);
        printf("Carry: %d\n",SREG & 1);
        printf("Ciphertext: ");
        for(i = 0; i < BLOCK_SIZE; i++)
        {
                printf("%02x ",state[i]);
        }
        printf("\n");

        sleep_cpu();

        return 0;
}
```

```
felics@felicsVM:~/Documents/SPECK$ simavr -f 8000000 -m atmega128 test.hex
Loaded 1 section of ihex
Load HEX flash 00000000, 3116
Carry: 0..
Ciphertext: 58 1f 25 4f a2 42 25 6f ..
```

bit-flips in the carry flag during the left circular shift operation and rightly guesses the outgoing carry of the modular addition. This makes the reference implementation [210] vulnerable, which was secure according to AA1. Nevertheless, going by AA2, the key expansion phase of SPECK cannot be attacked as l_{r+1} is unknown (see Figure 4.4).

Since SPECK$_{m,n}$ is intended to be implemented on 8-, 16- and 32-bit microcontrollers [212], the operation $\boxplus_{(\frac{m}{2})}$ will be performed using 8-, 16- and 32-bit additions, respectively. To perform the operation $\boxplus_{(\frac{m}{2})}$ on an ℓ-bit microcontroller, $\frac{m}{2\ell}$ ℓ-bit additions, starting from the ℓ least significant bits to the ℓ most significant bits, are required. The final addition will not set the carry flag unless $\frac{m}{2}$ is a multiple of ℓ. Therefore, the assumptions used in this Chapter will be valid only when SPECK$_{m,n}$ is implemented on an ℓ-bit microcontroller where $\frac{m}{2}$ is a multiple of ℓ. The SPECK implementations which

are vulnerable to our attacks are listed in Table 4.11. The possibility of implementing $\text{SPECK}_{m,n}$ in 64-bit platforms also has been discussed by the designers [212]; in such cases, if 64-bit accumulators are used, ciphers with a block size of 128 bits alone are vulnerable. If the flash memory is large enough to store the round keys, $\text{SPECK}_{m,n}$ can be implemented without the key expansion phase as suggested by the designers in [210]. In such implementations, our attack on the key schedule as given in Sect. 4.5 will not work.

Table 4.11: Vulnerability of $\text{SPECK}_{m,n}$ when implemented on ℓ-bit microcontroller, where $\ell = 8, 16$ or 32

Cipher	$\ell = 8$	$\ell = 16$	$\ell = 32$
$\text{SPECK}_{32,64}$	Yes	Yes	No
$\text{SPECK}_{48,72}$	Yes	No	No
$\text{SPECK}_{48,96}$	Yes	No	No
$\text{SPECK}_{64,96}$	Yes	Yes	Yes
$\text{SPECK}_{64,128}$	Yes	Yes	Yes
$\text{SPECK}_{96,96}$	Yes	Yes	No
$\text{SPECK}_{96,144}$	Yes	Yes	No
$\text{SPECK}_{128,128}$	Yes	Yes	Yes
$\text{SPECK}_{128,192}$	Yes	Yes	Yes
$\text{SPECK}_{128,256}$	Yes	Yes	Yes

Detecting the carry flag. The success of our attack solely depends on the chances of the adversary to read the carry flag at the end of a key expansion or encryption phase, if we are going by AA1. A complete treatment of the side-channel techniques used to obtain the carry flag is beyond the scope of this work. Nevertheless, we would like to highlight the following methods. An adversary who has the necessary privileges to execute any code of his choice in the encryption device will be able to detect the carry flag by executing the add with carry (ADC) instruction [213] soon after the key expansion or encryption phase. Return-oriented programming is a possible technique that can be used to execute a chosen instruction in the encryption device [214, 215, 216]. The only challenge will be to execute the code in the same core where the encryption program runs,

if it is a multicore processing device, and before the carry flag is potentially affected by any other operation.

Another possible method is to exploit the electromagnetic side-channel of the encryption device to detect the carry flag. In [217], Heyszl et al. presented a technique to distinguish the activities of registers by precisely measuring the electromagnetic field of small regions on integrated circuits using high resolution inductive probes. Also, it is practically feasible to distinguish between $0 \rightarrow 1$ and $1 \rightarrow 0$ bit transitions in certain implementations using electromagnetic analysis [218, 219]. Therefore, the attacker can analyse the electromagnetic field of the flag register to detect if the carry flag got set or reset during modular addition.

In [220], Fouque et al. demonstrated an attack on an implementation of HMAC-SHA-1 for a NIOS processor, where the electromagnetic side-channel was used to measure the number of bits flipped in a register. The same technique when applied on the flag register enables the attacker to detect the bit-flips in the carry flag, which validates AA2. Yet another method is to inject a fault during the computation of the carry, as explained by Fouque et al. [132] to get the carry information.

Improving the attack. Our work is originally motivated by the following remark due to Kelsey et al. [129]: “There may be cases when we can learn the state of the overflow or carry flag during encryption; if so, this can form a useful side-channel. [I]t demonstrates a useful feature of side-channel cryptanalysis: the side-channel can give information about the operations performed instead of the values used. This can work even when the information about the operation yields only minimal information about the values themselves.” It is possible that the attacker has access to other flags too. Overflow and parity flags, in some cases, help reduce the time complexities of our attacks by a factor of four.[9]

[9] In an ℓ-bit accumulator, a carry in the $(\ell - 1)$th bit sets overflow flag. Carry flag is set when there is an outgoing carry at the MSB position whereas overflow flag is set when there is an incoming carry at the MSB position. Parity flag indicates if the number of ones in the binary representation of the result of the last operation is odd or even.

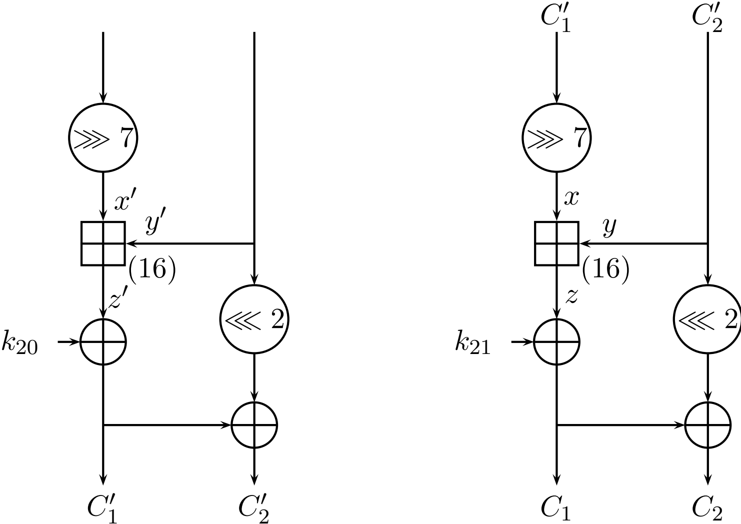


Figure 4.6: Last two rounds of encryption of SPECK$_{32,64}$

Another way to improve our attacks is to involve the penultimate round(s). For instance, let us assume that an unprotected implementation of SPECK$_{32,64}$ uses an 8-bit accumulator and the attacker can investigate the four additions from last two rounds. The two penultimate rounds of SPECK$_{32,64}$ are shown in Figure 4.6. Let x and y be the inputs to the modular addition in the final round and z be its output. As explained in Sect. 4.4, by observing 2^{14} encryptions and the corresponding carries generated from $F_0(x) + F_0(y)$ and $F_1(x) + F_1(y)$, five most significant bits of $F_0(k_{21})$ and $F_1(k_{21})$ can be recovered with 99.99% success rate.

Using $z = C_1 \oplus k_{21}$, five most significant bits of $F_0(z)$ and $F_1(z)$ can be computed and y is given by $(C_1 \oplus C_2) \ggg 2$. Let $x_{u,5}$, $y_{u,5}$ and $z_{u,5}$ be the five most significant bits of $F_1(x)$, $F_1(y)$ and $F_1(z)$, respectively. If $\widehat{x}_{u,5}$ denotes $(c_{out} \parallel z_{u,5}) - y_{u,5}$ then $x_{u,5}$ equals $\widehat{x}_{u,5} + c_{in}$, where c_{out} and c_{in} are the carries generated at the 8th and 3rd bit positions of $x_u + y_u$, respectively. The four most significant bits of $\widehat{x}_{u,5}$ and $x_{u,5}$ will be equal when the least significant bit of $\widehat{x}_{u,5}$ is 0. Since this occurs with probability 0.5, under the assumption that $\widehat{x}_{u,5}$ is distributed uniformly at random, in 2^{13} out of the 2^{14} encryptions observed, we can compute the four most significant bits of $x_{u,5}$ with 99.99% success rate. Since $C_1' = x \lll 7$, where $C_1' \parallel C_2'$ is the output of the 21st

round of SPECK$_{32,64}$, we get the four bits of $F_0(C'_1)$ corresponding to the known bits of $x_{u,5}$. If V represents the known bits and U represents the unknown bits, $F_0(C'_1)$ can be represented as $UVVVVUUU$ in 2^{13} encryptions. In each of these encryptions, if the carry flag after the first addition of the 21st round is known then the three bits of $F_0(k_{20})$ corresponding to the three most significant known bits of $F_0(C'_1)$ can be recovered with 99.99% success rate.[10]

Similar analysis can be used to recover 3 bits of $F_1(k_{20})$. Thus 16 bits of the 64-bit key can be recovered with 99.99% success rate by observing 2^{14} encryptions. When the overflow flag information is also exploited, 20 bits of the key can be recovered with the same success rate and complexity.

Countermeasure. A simple method to preclude our attack is to mask the carries generated at the end of the key expansion and encryption phases by introducing low-cost arithmetic operations after step 8 and step 7 of Algorithms 8 and 9, respectively, which permanently set or clear the carry flag. However, the approach fails if the attack model assumes that the attacker can determine the carry flag at any stage of execution of the code or skip the arithmetic operation introduced to mask the outgoing carry of the modular addition.

4.7 Conclusions

In this Chapter, we have shown side-channel attacks on the SPECK family of block ciphers which is an ISO/IEC standard for RFID devices. Our attacks are applicable on the full ciphers. Although there are a few fault attacks which work on the full SPECK, our attacks are comparatively more feasible due to the weaker assumptions we make. The data requirements for our attacks are well below the respective birthday bounds. Depending on the key size and block size, the complexities of our attacks, to nearly

[10]By observing 2^{13} encryptions, at the most four bits of $F_0(k_{20})$ can be recovered with 99.99% success rate, given that the four most significant bits of $F_0(C'_1)$ are known.

ensure success, vary from 2^{59} time and 2^{14} data to 2^{227} time and 2^{62} data. Since the vulnerability of the cipher to our attack depends on the way it is implemented, the details of the vulnerable implementations have also been provided. We have also proposed a countermeasure to preclude our attacks.

Chapter 5

Fault-Assisted Side-Channel Analysis of HMAC-Streebog

5.1 Introduction

HMAC-Streebog. Streebog is a family of hash functions developed by the Center for Information Protection and Special Communications of the Federal Security Service of the Russian Federation with the participation of the open joint-stock company Information Technologies and Communication Systems (InfoTeCS JSC) [150]. It is defined in the Russian cryptographic standard GOST R 34.11–2012 [150]. Streebog is comprised of two hash functions, Streebog-256 and Streebog-512, which generate 256-bit and 512-bit message digests, respectively.

HMAC is an algorithm to calculate a message authentication code (MAC), based on a hash function [221]. In its first phase, an inner hash is derived from the message and the inner key. Later, it generates the outer hash from the inner hash and the outer key, and outputs it as the MAC. The specifications of the HMAC algorithms based on Streebog has been defined in RFC 7836 [144]. Corresponding to Streebog-256 and Streebog-512, two HMAC algorithms exist which we call HMAC-Streebog-256 and HMAC-Streebog-512, respectively. The recommended key sizes of HMAC-Streebog-

256 and HMAC-Streebog-512 are at least 256 bits and 512 bits, respectively, and not more than 512 bits [221].

Considering its significance, Streebog was well studied over a period of time [222, 223, 224, 225, 226, 227, 131]. Except for the side-channel attack by Sekar [131] and the differential fault attack by AlTawy et al. [227], attacks on Streebog are on its reduced-round variants. In [228], Dinur et al. presented the first key recovery attack on HMAC-Streebog-512 with a complexity of 2^{410}. Later, AlTawy et al. presented differential fault attacks [227] on HMAC-Streebog-512 and HMAC-Streebog-256 which can recover the key by injecting single-bit faults into the last two intermediate states of all the compression functions of the outer hash function. The authors claimed that the average number of faults required to recover the input of each compression function of Streebog vary between 338 and 1640.

Contributions of this work. Side-channel attacks on HMAC-Streebog are presented in this Chapter. Carry flag is a bit of the status register, present in nearly every modern microprocessor, that indicate carry overflow in unsigned integer arithmetic. In [131], Sekar conjectured that the carry flag based side-channel attack could speed up the key recovery of HMAC-Streebog-256 and HMAC-Streebog-512 by a factor of 2, in certain cases. Our investigation of this conjecture resulted in passive side-channel attacks on HMAC-Streebog-256 and HMAC-Streebog-512 which can recover one bit of the respective key with a success rate of 75%. The attacks work under the assumption that the inner hash of the HMAC and the carry flag at the end of MAC generation are known to the attacker. We find that the attacks can be further improved by injecting faults into the output of the inner hash function and propose active attacks on HMAC-Streebog-256 and HMAC-Streebog-512 which can recover the keys with $2^{12.98}$ and $2^{14.97}$ average number of fault injections, respectively, for 95% success rate. We assume that the attacker is able to flip at the most 181 and 361 chosen bits at a time, respectively, for HMAC-Streebog-256 and HMAC-Streebog-512.

To the best of our knowledge, our passive attack is the best non-fault attack on HMAC-Streebog-256. Compared to the attack in [227], our attacks have a longer temporal window for fault injection and cannot be mitigated using output redundancy countermeasures as we rely on the carry flag side-channel.[1,2] Also, it is reasonable to assume that the inner hash being an intermediate output is more accessible than the internal state of Streebog. If the intermediate carries generated by the ripple carry adder implementation of the targeted addition are known, complexities of our passive attacks and fault requirements of our active attacks reduce as each adder can be attacked separately. When implemented in systems which are vulnerable to attacks like Rowhammer [230], Meltdown [125], Spectre [126], SplitSpectre [127], RAMBleed [128] and Cold Boot [231], HMAC-Streebog will be highly vulnerable to our attacks.

Organisation of the Chapter. The remaining Chapter is organised as follows. Section 5.2 describes Streebog and HMAC-Streebog. We present our motivational observations in Sect. 5.3. Our attacks on HMAC-Streebog-512 and HMAC-Streebog-256 are presented in Sect. 5.4. The validity of our assumptions is discussed in Sect. 5.5 and we conclude in Sect. 5.6.

5.2 Specification of HMAC-Streebog

5.2.1 Description of Streebog

Streebog accepts any message M of length less than 2^{512} bits and returns a 256- or 512-bit digest. If the message length $|M|$ is not a multiple of 512, M is prefixed with a bit string $pad := \{0\}^{511-(|M| \bmod 512)} \parallel 1$. The padded message is then partitioned into $k+1$ 512-bit blocks $M_k, M_{k-1}, \ldots, M_0$ such that $pad \parallel M = M_k \parallel M_{k-1} \parallel \ldots \parallel M_0$.

[1] The data to be altered is available for more than $2t$ time, where t is the time taken by the compression function of Streebog, as the targeted modular addition is executed after two compression operations.

[2] In output redundancy countermeasures, data is processed via redundant channels and the output will not be generated unless all of them agree to it. Still, the carry flag side-channel remains unaffected.

Algorithm 13 The Streebog algorithm

Require: The message M, $|M| < 2^{512}$
Ensure: A 256-bit or a 512-bit digest

1: $M \to pad \parallel M \to M_k \parallel M_{k\text{-}1} \parallel \ldots \parallel M_0$;
2: $\pi_0 = IV$; $n_0 = 0$; $S = 0$;
3: **for** $i = 0$ to $(k-1)$ **do**
4: $\quad \pi_{i+1} = G(\pi_i, M_i, n_i)$;
5: $\quad n_{i+1} = n_i + 512 \bmod 2^{512}$;
6: $\quad S = S + M_i \bmod 2^{512}$;
7: **endfor**
8: $\pi_{k+1} = G(\pi_k, M_k, n_k)$;
9: $n_{k+1} = n_k + 512 - |pad| \bmod 2^{512}$;
10: $S = S + M_k \bmod 2^{512}$;
11: $\pi_{k+2} = G(\pi_{k+1}, n_{k+1}, 0)$;
12: $H = G(\pi_{k+2}, S, 0)$;
13: Output H if Streebog-512, else output $H \gg 256$;

The compression function G has 13 iterations out of which 12 involve a substitution-permutation layer which consists of the following components: a substitution function Γ, a permutation function $\mathcal{F}$, a linear transformation $\mathcal{L}$ and a function $\mathcal{X}[\cdot]$. The substitution function substitutes each byte of its 512-bit input by a byte from a permuted set of $\{0, 1, \ldots, 255\}$ and the permutation function shuffles the position of each byte in its 512-bit input. The linear transformation of a 512-bit input W is performed as follows: $\mathcal{L}(W) = l(\Psi_7(W)) \parallel l(\Psi_6(W)) \parallel \ldots \parallel l(\Psi_0(W))$, where l outputs the right multiplication of its input with a constant matrix **A** over $GF(2)$ (see (5.1)). If a and b are 512-bit strings, $\mathcal{X}[a](b) = a \oplus b$. The compression function G that processes the message block M_i takes as additional inputs the 512-bit chaining value π_i and a length counter n_i, and outputs π_{i+1} (see (5.3)). The initial value π_0 is a 512-bit IV which is different for Streebog-256 and Streebog-512. Algorithm 13 describes the working of Streebog.

$$l(\zeta) = \bigoplus_{i=0}^{63} \zeta_{(63-i)} \odot \mathbf{A}[i]\,, \tag{5.1}$$

where

$$\zeta_{(63-i)} \odot \mathbf{A}[i] = \begin{cases} \{0\}^{64},\ \text{if}\ \zeta_{(63-i)} = 0\,; \\ \mathbf{A}[i],\ \text{if}\ \zeta_{(63-i)} = 1\,. \end{cases} \tag{5.2}$$

$$\pi_{i+1} := G(\pi_i, M_i, n_i) = E(\mathcal{L}(\mathcal{F}(\Gamma(\pi_i \oplus n_i))), M_i) \oplus \pi_i \oplus M_i\,, \tag{5.3}$$

where

$$E(\mathcal{L}(\mathcal{F}(\Gamma(\pi_i \oplus n_i))), M_i) = \mathcal{X}[\nu_{13}]\mathcal{L}\mathcal{F}\Gamma\mathcal{X}[\nu_{12}]\mathcal{L}\mathcal{F}\Gamma\mathcal{X}[\nu_{11}]\ldots\mathcal{L}\mathcal{F}\Gamma\mathcal{X}[\nu_1](M_i)\,, \tag{5.4}$$

and $\nu_1,\ \nu_2,\ \ldots,\ \nu_{13}$ are derived as,

$$\nu_0 = \mathcal{L}\mathcal{F}\Gamma(\pi_i \oplus n_i)\,, \tag{5.5}$$

$$\nu_{j+1} = \mathcal{L}\mathcal{F}\Gamma(\nu_j \oplus C_j)\,,\ \text{for}\ j = 0,\ 1,\ \ldots,\ 12,\ \text{and constants}\ C_j. \tag{5.6}$$

5.2.2 Description of HMAC-Streebog

A HMAC algorithm employs a hash function h in conjunction with a secret key K and generates a MAC value as follows:

$$\text{HMAC}(K, M) = h\Big((K_0 \oplus opad), h((K_0 \oplus ipad), M)\Big)\,, \tag{5.7}$$

where M is the message, $opad$ and $ipad$ are public constants, and K_0 is the secret key if $|K|$ equals the block size of h, or a function of K otherwise. HMAC-Streebog will use either Streebog-512 or Streebog-256 as the underlying hash function. According to RFC 2104 [221], the recommended length of the secret key for a secure HMAC is equal to the digest length of h, and it cannot be more than the block size of h. Therefore the

key size of HMAC-Streebog has to be at least 256 or 512 bits depending on whether Streebog-256 or Streebog-512 is used as the hash function. If K is shorter than the block size, it is padded with $\{0\}^{512-|K|}$ to get K_0. Algorithm 14 describes the working of HMAC-Streebog, where h is Streebog-256 or Streebog-512.

Algorithm 14 The HMAC-Streebog algorithm

Require: The message M and secret key K in big-endian format
Ensure: A 256-bit or a 512-bit digest

1: $K_0 = \{0\}^{512-|k|} \parallel K$;
2: $ipad = \{00110110\}^{64}$;
3: $opad = \{01011100\}^{64}$;
4: $H_{in} = h(M \parallel (K_0 \oplus ipad))$;
5: $H_{out} = h(H_{in} \parallel (K_0 \oplus opad))$;
6: Output H_{out};

5.3 Motivational Observations

5.3.1 On Carry in Modular Addition

Table 5.1: Table showing the allowed values of the Boolean variables $x_{(i-1)}, y_{(i-1)}$, $c_{(i-1)}$, $c_{(i)}$ and $c'_{(i)}$, and the probability of each event where $p_i = \Pr(c_{(i)} = 0)$, for $i > 1$

$x_{(i-1)}$	$y_{(i-1)}$	$c_{(i-1)}$	$c_{(i)}$	$c'_{(i)}$	**Probability**
0	0	0	0	0	$\frac{1}{4}p_{i-1}$
0	0	1	0	1	$\frac{1}{4}(1-p_{i-1})$
0	1	0	0	1	$\frac{1}{4}p_{i-1}$
0	1	1	1	1	$\frac{1}{4}(1-p_{i-1})$
1	0	0	0	0	$\frac{1}{4}p_{i-1}$
1	0	1	1	0	$\frac{1}{4}(1-p_{i-1})$
1	1	0	1	0	$\frac{1}{4}p_{i-1}$
1	1	1	1	1	$\frac{1}{4}(1-p_{i-1})$

Let x and y, which are distributed uniformly at random, be the inputs to an n-bit modular addition, $c_{(i)}$ be the carry generated at the ith bit position of the addition, where $1 \leq i \leq n$ (for instance, outgoing carry at the LSB position is denoted by $c_{(1)}$), and c_{out} be the final carry which equals $c_{(n)}$. The carries generated at the ith bit position when input x is replaced with $x \oplus I_{n,i-1}$ is denoted by $c'_{(i)}$. The allowed values of the Boolean variables $x_{(i-1)}$, $y_{(i-1)}$, $c_{(i-1)}$, $c_{(i)}$ and $c'_{(i)}$, for $i > 1$ and the probability of each event (each possibility is considered an event) are tabulated in Table 5.1.[3] Let us consider two cases.

Case 1: The ith bit of x and $c_{(i)}$ for any $i \in \{n, n-1, \ldots, 1\}$ are known.
From Table 5.1, we can deduce the following.
If $x_{(i-1)} \oplus c_{(i)} = 1$,

$$y_{(i-1)} = c_{(i)}, \; n \geq i \geq 2\,, \tag{5.8}$$

$$c_{(i-1)} = c_{(i)}, \; n \geq i \geq 2\,. \tag{5.9}$$

Else, assuming the events given in Table 5.1 are mutually exclusive,

$$\Pr(y_{(i-1)} = c_{(i)}) = \frac{2}{3}, \; n \geq i \geq 2\,. \tag{5.10}$$

Since $c_{(0)} = 0$,

$$y_{(0)} = c_{(1)} \iff x_{(0)} \oplus c_{(1)} = 1\,, \tag{5.11}$$

$$y_{(0)} = 1 \iff x_{(0)} = c_{(1)} = 1\,. \tag{5.12}$$

[3] We assume that $x_{(i-1)}$, $y_{(i-1)}$ and $c_{(i-1)}$ are independent, and $\Pr(x_{(i-1)} = 0) = \Pr(y_{(i-1)} = 0) = 0.5$.

Case 2: The ith bit of x, $c_{(i)}$ and $c'_{(i)}$ for any $i \in \{n, n-1, \ldots, 1\}$ are known.

As in Case 1, equations (5.8), (5.9), (5.11) and (5.12) hold, if the respective conditions are satisfied. In addition to them, we get the following.

If $x_{(i-1)} \oplus c_{(i)} = 0$ and $c_{(i)} = c'_{(i)}$,

$$y_{(i-1)} = c_{(i)},\ n \geq i \geq 1\,, \tag{5.13}$$

$$c_{(i-1)} = c_{(i)},\ n \geq i \geq 2\,. \tag{5.14}$$

If $x_{(i-1)} \oplus c_{(i)} = 0$ and $c_{(i)} \neq c'_{(i)}$,

$$y_{(i-1)} = c_{(i-1)} \oplus 1,\ n \geq i \geq 2\,, \tag{5.15}$$

$$y_{(i-1)} = 1,\ \text{for } i = 1\,. \tag{5.16}$$

5.3.2 On the Recovery of an Unknown Operand

Let $f^n_y(x)$ be a function which takes an n-bit secret input y, which is distributed uniformly at random, and an n-bit known input x, which is generated by a random oracle,[4] and outputs $z := (x + y) \bmod 2^n$. If the carry c_{out} is known, based on the observations given in Sect. 5.3.1, the secret input y can be recovered quicker than exhaustive search. We describe two methods to recover y, one each for the Cases of Sect. 5.3.1.

Passive analysis. Based on Case 1, a passive analysis can be performed to recover the k most significant bits of y using (5.8)–(5.12), if c_{out} and the k most significant bits of x are known.

[4] A random oracle is a theoretical black box that responds to every unique query with a response chosen uniformly at random from its output domain.

Algorithm 15 Recovering the secret input y of $f_y^n(x)$ using active analysis

Require: Known input x, carry c_{out} and carries generated when x is modified
Ensure: n-bit y

1: $Y = \{0\}; C = \{c_{out}\}$;
2: **for** $i = n$ to 1 **do**
3: $Ynew = \{\}$; $Cnew = \{\}$; $cnt = 0$;
4: **for** $j = 0$ to **sizeof**$(Y) - 1$ **do**
5: $y \leftarrow Y[j]$; $c_{(i)} \leftarrow C[j]$;
6: **if** $x_{(i-1)} \oplus c_{(i)} = 1$
7: $y_{(i-1)} = c_{(i)}$;
8: **if** $i \neq 1$
9: $c_{(i-1)} = c_{(i)}$;
10: **endif**
11: **else if** $i = 1$ & $c_{(1)} = 1$
12: $y_{(0)} = 1$;
13: **else**
14: $x' = x \oplus I_{n,i-1}$;
15: **if** $i \neq n$
16: $x'_l = x'_{(i-1)} \parallel x'_{(i-2)} \dots \parallel x'_{(0)}$;
17: $y_u = y_{(n-1)} \parallel y_{(n-2)} \dots \parallel y_{(i)}$;
18: $y'_u = y_u \oplus (2^{n-i} - 1)$;
19: $c'_{(i)}$ = carry generated after executing $f_y^n(y'_u \parallel x'_l)$;[a]
20: **else**
21: $c'_{(i)}$ = carry generated after executing $f_y^n(x')$;
22: **endif**
23: **if** $c'_{(i)} = c_{(i)}$
24: $y_{(i-1)} = c_{(i)}$;
25: **if** $i \neq 1$
26: $c_{(i-1)} = c_{(i)}$;
27: **endif**
28: **else**
29: $y_{(i-1)} = 1$;
30: **if** $i \neq 1$
31: $Ynew[cnt] = y$;
32: $Ynew[cnt]_{(i-1)} = 0$;
33: $Cnew[cnt] = 1$;
34: $c_{(i-1)} = 0$;
35: $cnt = cnt + 1$;
36: **endif**
37: **endif**
38: **endif**
39: $Y[j] \leftarrow y$; $C[j] \leftarrow c_{(i-1)}$;
40: **endfor**
41: **for** $j = 0$ to $cnt - 1$ **do**
42: $Y[j+$**sizeof**$(Y)] = Ynew[j]$;
43: $C[j+$**sizeof**$(Y)] = Cnew[j]$;
44: **endfor**
45: **endfor**
46: Choose y from Y using exhaustive search
47: Output y

[a] Since $y'_u + y_u = \{1\}^{n-i}$, carry generated at the ith bit position of $(y'_u \parallel x'_l) + y$ will be equal to that at the MSB.

Active analysis. From Case 2, Algorithm 15 has been derived and it additionally re-

quires the carries generated when a set of chosen x's are given as inputs to $f_y^n(x)$. Chosen x's are used in steps 19 and 21.[5]

5.3.3 Passive Analysis vs Active Analysis

According to Case 1 of Sect. 5.3.1, $x_{(i-1)} \oplus c_{(i)}$ and $x_{(i-2)} \oplus c_{(i-1)}$ equal 1, if and only if $x_{(i-1)}$ and $x_{(i-2)}$ are equal, for $i > 1$. Hence, to recover the k most significant bits of y, where $k \neq n$, with 100% success rate, either c_{out} must be 1 and the corresponding bits of x must be 0 or vice versa; probability of this event turns out to be 2^{-2k}. Therefore, only a few bits can be recovered with this method. Nevertheless, the most significant bit of y can be recovered with 75% success rate using passive analysis as it can be recovered with $\frac{2}{3}$ probability when $x_{(n-1)} \oplus c_{out}$ equals 0.[6]

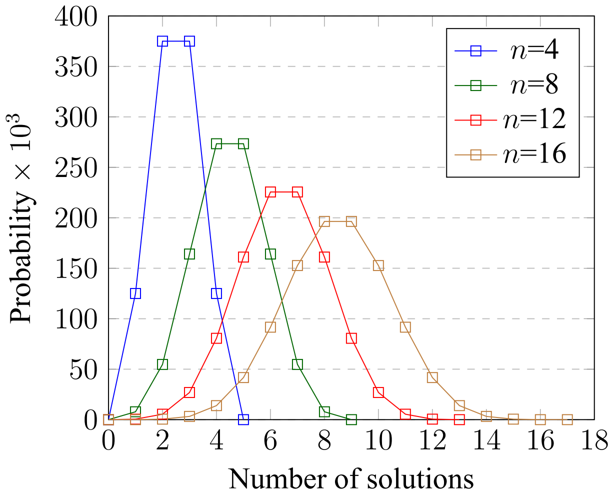


Figure 5.1: Probability distributions of the size of Algorithm 15's solution space for $n =$ 4, 8, 12 and 16

Compared to passive analysis, active analysis can recover more number of bits with a better success rate. At steps 29 and 31–34, $(y_{(i-1)}, c_{(i-1)})$ takes the values $(0, 1)$ and $(1, 0)$, following (5.15), due to which a set of possible solutions of y gets generated. Instead of performing an exhaustive search over the entire space, our search is limited

[5] Since x and y are independent and distributed uniformly at random, $y'_u \parallel x'_l$ and x', which constitute the chosen x's, are also uniformly distributed. The definitions of y'_u, x'_l and x' are given in Algorithm 15.

[6] Success probability $:= \frac{1}{4} \cdot 1 + \frac{3}{4} \cdot \frac{2}{3} = 0.75$.

to this set of solutions. In order to analyse the size of this solution space, the algorithm was tested with all possible values of x and y for $n = 4$, 8, 12 and 16. The results show that for any n, at the most n, and on an average $\frac{n}{2}$ or $\frac{n}{2}+1$, solutions will be generated. The probability distributions of the size of solution space for $n = 4$, 8, 12 and 16 are shown in Figure 5.1.

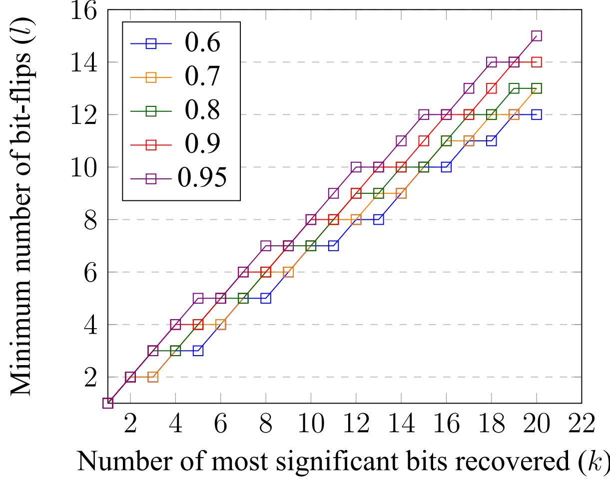


Figure 5.2: Minimum number of simultaneous bit-flips (l) required to recover the k most significant bits for each of the success probabilities 0.6, 0.7, 0.8, 0.9 and 0.95

The other factors affecting the active analysis are the maximum number of modifications of x required and the maximum number of simultaneous bit-flips needed at any instance of modification. Knowing $x_{(i-1)}$ and $c_{(i)}$, probabilities of passively recovering $y_{(i-1)}$ with 100% success rate for $i \neq 1$ and $i = 1$ are 0.25 and 0.5, respectively. Hence, to recover k most significant bits of y, x has to be modified at the least $0.75k$ times. When multiple solutions of y are generated, the number of modifications of x will be more than $0.75k$ as it depends on the existing solutions of y (see step 19 of Algorithm 15). The average number of modifications of x required to recover the ith bit of y was experimentally computed to be equal to $\frac{3}{4} + \frac{1}{4}\sum_{j=0}^{n-i}(1 - 2^{-j})$ where $i \in \{n, n-1, \ldots, 1\}$. In order to understand the effect of the number of simultaneous bit-flips on the number of bits recovered, we performed another test using all possible inputs for $n = 20$. The minimum number of simultaneous bit-flips l required to recover the k most significant bits of y for each of the success probabilities 0.6, 0.7, 0.8, 0.9 and 0.95 where

experimentally computed, and the results are plotted in Figure 5.2. The linear equations $0.6k-l+0.5=0, 0.626k-l+0.574=0, 0.636k-l+0.821=0, 0.677k-l+0.837=0$ and $0.716k-l+0.884=0$, which were inferred using curve fitting techniques, define the relationship between l and k for the success probabilities 0.6, 0.7, 0.8, 0.9 and 0.95, respectively.

5.4 Key Recovery Attacks on HMAC-Streebog

We shall now discuss how an attacker, who has access to the authentication device of HMAC-Streebog-512, can recover K with a complexity lesser than that of exhaustive key search. We make the following assumptions:

1. The attacker knows the inner hash H_{in} mentioned in Algorithm 14.
2. The attacker is able to detect the carry flag at the end of message authentication code generation.
3. The attacker is able to alter chosen bits of H_{in} on demand.

From step 5 of Algorithm 14 and step 10 of Algorithm 13, we see that the final operation of HMAC-Streebog-512 that affects the carry flag is $H_{in}+K' \bmod 2^{512}$, where $K' = K_0 \oplus opad$. This is because the functions Γ, $\mathcal{F}$, $\mathcal{X}[\cdot]$ and $\mathcal{L}$ do not affect the carry flag. Consequently, the compression function G does not affect the carry flag. We assume that the machine code generated from the reference C implementation of Streebog [229] does not contain any additional operation that affects the carry flag after the final modular addition.

If H_{in} and K' are considered to be x and y, respectively, then $H_{in} + K' \bmod 2^{512}$ equals $f_y^{512}(x)$, where $f_y^n(x)$ is the function defined in Sect. 5.3.2. Since H_{in} and the carry generated by $f_y^{512}(x)$ are known to the attacker, he may be able to recover some of the most significant bits of K' using passive analysis. The strength of the attacker to alter the chosen bits of H_{in} enables him to perform active analysis to recover the entire K' and, in turn, the knowledge of K' leads to the recovery of entire K.

5.4.1 Complexities of the Attacks

Table 5.2: Variation of the number of key bits recovered ($i \cdot l$) using Algorithm 15 and the required number of modifications of H_{in} in logarithmic scale ($\log_2 \tau_{i \cdot l}$) with the possible number of chosen bit-flips (l) where $i = 1.4,\ 1.48,\ 1.57,\ 1.6,\ 1.67$ and $i \cdot l \leq 512$

l	$1.4l$	$\log_2 \tau_{1.4l}$	$1.48l$	$\log_2 \tau_{1.48l}$	$1.57l$	$\log_2 \tau_{1.57l}$	$1.6l$	$\log_2 \tau_{1.6l}$	$1.67l$	$\log_2 \tau_{1.67l}$
1	1	-0.42	1	-0.42	1	-0.42	1	-0.42	1	-0.42
31	43	7.95	45	8.08	48	8.26	49	8.32	51	8.43
61	85	9.87	90	10.03	95	10.19	97	10.24	101	10.36
91	127	11.01	134	11.16	142	11.33	145	11.39	151	11.51
121	169	11.83	179	11.99	189	12.15	193	12.21	202	12.34
151	211	12.46	223	12.62	237	12.80	241	12.84	252	12.97
181	253	12.98	267	13.14	284	13.31	289	13.36	302	13.49
211	295	13.42	312	13.58	331	13.75	337	13.81	352	13.93
241	337	13.81	356	13.96	378	14.14	385	14.19	402	14.31
271	379	14.14	401	14.31	425	14.47	433	14.53	452	14.65
301	421	14.45	445	14.61	472	14.77	481	14.83	502	14.95
331	463	14.72	489	14.88	-	-	-	-	-	
361	505	14.97	-	-	-	-	-	-	-	-

Knowing H_{in} and carry flag, the most significant bit of K can be recovered using passive analysis with 75% success rate. In other words, the time complexity of the key recovery attack using passive analysis for 75% success rate will be $O(2^{511})$. The complexity of recovering K using active analysis varies depending on the strength of the attacker. In Sect. 5.3.3, we have already discussed the average number of modifications and simultaneous bit-flips required to recover the bits of the unknown variable y. In order to recover the k most significant bits of K, H_{in} has to be modified τ_k times on an average, where τ_k can be calculated as:

$$\tau_k = \sum_{i=512}^{512-k+1} \left(\frac{3}{4} + \frac{1}{4} \sum_{j=0}^{512-i} (1 - 2^{-j}) \right) .$$

At least $0.716k$ simultaneous bit-flips are required for each modification to ensure a 95% success. Therefore, if the attacker has enough resources to flip l chosen bits of H_{in} simultaneously, $1.4l$ most significant bits of K can be recovered using Algorithm 15 with 0.95 success probability and, he can recover the remaining bits through brute force. The time complexity to recover K using active analysis will be $\chi(l) + O(2^{(512-1.4l)})$ for a success rate of 95%, where $\chi(l)$ is the time required for $\tau_{1.4l}$ modifications of H_{in}.

Assuming that each modification takes $O(\alpha)$ time, $\chi(l) = O(\alpha \cdot \tau_{1.4l})$. Therefore the complexity of our active attack for a success rate of 95% will be $O(\alpha \cdot \tau_{1.4l} + 2^{(512-1.4l)})$, which equals $O(2^{(512-1.4l)})$ if $\log_2 \alpha \cdot \tau_{1.4l} \ll 512 - 1.4l$. Similarly, the respective complexities of our attacks for the success rates of 90%, 80%, 70% and 60% will be $O(\alpha \cdot \tau_{1.48l} + 2^{(512-1.48l)})$, $O(\alpha \cdot \tau_{1.57l} + 2^{(512-1.57l)})$, $O(\alpha \cdot \tau_{1.6l} + 2^{(512-1.6l)})$ and $O(\alpha \cdot \tau_{1.67l} + 2^{(512-1.67l)})$. The variation of $i \cdot l$ and $\log \tau_{i \cdot l}$ with l, where $i = 1.4,\ 1.48,\ 1.57,\ 1.6,\ 1.67$, is listed in Table 5.2.

For HMAC-Streebog-256, if $|K| = 256$, the 256 most significant bits of K' and $pad \parallel H_{in}$ will be known to the attacker as they are the padding bits. Therefore, he will be able to compute $c_{(256)}$ from $c_{(512)}$, where $c_{(512)}$ is the carry flag. Hence the attack on 512-bit addition reduces to that on 256-bit addition, which will, in turn, result in the reduction of the attack complexity. For instance, the complexities of our passive attack with success rate of 75% and active attack with success rate of 95% will reduce to $O(2^{255})$ and $O(\alpha \cdot \tau_{1.4l} + 2^{(256-1.4l)})$, respectively.

If the attacker is able to flip at the most 181 and 361 chosen bits of H_{in} then he will be able to recover the keys of HMAC-Streebog-256 and HMAC-Streebog-512, respectively, with negligible complexity. The average number of modifications of H_{in} required for the attacks on HMAC-Streebog-256 and HMAC-Streebog-512 are $2^{12.98}$ and $2^{14.97}$, respectively.

5.5 Validity of the Assumptions

Our passive attack ceases to exist if any of the first two assumptions made in Sect. 5.4 is invalid. Similarly, key recovery using active analysis will be possible only if the three associated assumptions of Sect. 5.4 are valid. Therefore, it is necessary to examine the possibility of our assumptions in a real-world context.

5.5.1 Extraction of the Inner Hash H_{in}

In [125] and [126], Lipp et al. and Kocher et al. have respectively presented two attacks, the Meltdown attack and the Spectre attack, which enable a malicious application to access the memory space of a different application and read its secrets by exploiting critical vulnerabilities in modern processors. SplitSpectre [127], another speculative execution attack, is a recent variant of Spectre attack proposed by Mambretti et al. which requires a smaller piece of vulnerable code available in the victim's attack surface compared to the original attack. RAMBleed attack [128] by Kwong et al. is a more recent attack which exploits the Rowhammer vulnerability [230] in DRAM cells to read some of the bits in any DDR3 and DDR4 DRAM memories without accessing them. If the attacker has physical access to the hardware, Cold Boot attack [231] is yet another method to extract data from the RAM. The above mentioned attacks target a wide variety of platforms such as personal computers, mobile devices, embedded systems and even cloud environment. Since these attacks are based on hardware-related vulnerabilities, mitigating them without upgrading the hardware cannot be fully ensured. Moreover, efficient variants of the known attacks or even new line of attacks unveil in the course of time.

In order to extract H_{in} from the authentication device, the attacker can use an unprivileged spy software within the same device which utilises one of the above mentioned attacks to read the memory. Being a message digest, it is reasonable to assume that the extraction of H_{in} from memory will not be as difficult as that of a secret key. Though the secret key can be secured against such attacks by processing them outside the memory [232, 233, 234, 235, 236, 237], such protection mechanisms are not applicable to H_{in}.

5.5.2 Detection of Carry

The 512-bit addition in Streebog will be mostly implemented as a ripple carry adder where the word size of the full adders will be equal to or less than that of the accumulator of the underlying processor as in [229]. If they are equal, carry flag will be set

when a carry is generated. An attacker who has the necessary privileges to execute any code of his choice in the authentication device will be able to detect the carry flag by executing the add with carry (ADC) assembly instruction. If this method is not feasible, he can use techniques similar to those explained by Fouque et al. which will either exploit the electromagnetic side-channel of the device [133] or inject a fault during the computation of the carry [132] to get the carry information. It is easier to detect the carry flag soon after the MAC generation rather than while it is under generation. But, if the above mentioned techniques are performed in synchronisation with the intermediate additions which implement the targeted addition, the attacker will be able to detect all the intermediate carries which, in turn, will reduce the complexities of our attacks as each intermediate addition can be attacked separately.

If the word size of the adder is less than that of the accumulator, carry flag or overflow flag will never be affected. Nevertheless, the intermediate carries will be stored in the buffer as in [229] to forward them to the adjacent adders. The attacker can extract them from the buffer using one of the methods mentioned in Sect. 5.5.1.

5.5.3 Chosen Bit Modification of Inner Hash H_{in}

In [230], Kim et al. showed the Rowhammer vulnerability that causes bits in the rows adjacent to the frequently activated rows of a DRAM memory to flip without access. Cojocar et al. [238] demonstrated that multiple bits could be flipped in a chosen manner exploiting this vulnerability and the bits tend to flip to the same value of the bits of the corresponding column in adjacent rows. Very recently, Kwong et al. [128] described and demonstrated the steps to be followed to flip the bits of a target data in the DRAM memory. An unprivileged spy software, similar to that used for the extraction of H_{in}, can be used for its modification too, using the Rowhammer attack.

Another method to flip the bits of H_{in} is by injecting multiple single-bit faults using multi-spot lasers, as mentioned in [239]. But the number of simultaneous bit-flips induced utilising this method will be lesser compared to the earlier one, which leads to

an increased attack complexity. Still, if the word size of the adders used in the ripple carry adder is small, which will be the case when HMAC-Streebog is implemented in an embedded system, and the intermediate carries are known, the complexity of our attack can be reduced by applying Algorithm 15 on each adder seperately instead of 512-bit ripple carry adder.

5.6 Conclusions

In this Chapter, we have shown side-channel attacks on HMAC-Streebog which is a HMAC algorithm based on the Russian standard Streebog and defined in RFC 7836. Our attacks use carry flag as the side-channel to recover the keys with complexities lesser than that of exhaustive search. Under the assumption that the output of the inner hash function of HMAC-Streebog is known to the attacker, our passive side-channel attacks on HMAC-Streebog-256 and HMAC-Streebog-512 can recover one bit of the respective key with 75% success rate. We have also presented fault-assisted side-channel attacks on HMAC-Streebog-256 and HMAC-Streebog-512 which can recover the keys with $2^{12.98}$ and $2^{14.97}$ average number of fault injections, respectively, for 95% success rate under the assumption that the attacker is able to simultaneously flip at the most 181 chosen bits for HMAC-Streebog-256 and 361 chosen bits for HMAC-Streebog-512. We have highlighted some of the latest hardware vulnerabilities which make the HMAC-Streebog implementations vulnerable to our attacks. To the best of our knowledge, our passive attack is the best non-fault attack on HMAC-Streebog-256. Compared to the other fault attacks on HMAC-Streebog, our attacks have a larger temporal window for fault injection, target a more accessible location and cannot be mitigated with output redundancy countermeasures. The attacks presented here emphasises the importance of preventing the side-channel leakage of carry bits.

Chapter 6

Revisiting the Software-Efficient Stream Ciphers RCR-64 and RCR-32

6.1 Introduction

The Py family of stream ciphers. In 2005, the first members of the Py family of synchronous stream ciphers designed by Biham and Seberry — the Py and the Py6 — were submitted to the ECRYPT eSTREAM project in the category of software-based stream ciphers [25]. The internal states of the ciphers were primarily constituted by rolling arrays making them structurally similar to RC4 [21]. Owing to the impressive speed in software and being one of the fastest eSTREAM "Profile 1" candidates [153], the cipher Py gained much attention. In 2006, Paul et al. reported distinguishing attacks against Py [240], and its complexity was further reduced by Crowley [241]. To rule out these attacks, the designers proposed the cipher Pypy [152]. In 2007, Wu and Preneel showed key recovery attacks against the ciphers Py, Pypy and Py6 based on a weakness found in their identical IV setup algorithms [242]. In response to these attacks, which were later improved by Isobe et al. [243], the designers tweaked the IV setup of the ciphers leading to the design of TPy, TPypy and TPy6 [151].

The RCR ciphers. Immediately after the publication of TPy, TPypy and TPy6 in 2007, they were subjected to various distinguishing attacks [244, 245, 145, 246]. In a paper discussing related-key distinguishers on the ciphers TPy and TPypy, Sekar et al. presented the stream ciphers RCR-64 and RCR-32 by modifying TPy and TPypy, respectively [145]. The RCR-64 (resp. RCR-32) and the TPy (resp. TPypy) have identical key/IV setup algorithms. They differ in the round functions where variable rotations are replaced with constant rotations. Sekar et al. showed that these modifications not only eliminate the weaknesses of the ciphers TPy and TPypy but also make them marginally faster [145].

Motivation behind this work. The RCR ciphers are conjectured to be the strongest among the Py family of ciphers. In [146], Ding et al. reported distinguishing attacks on the RCR ciphers but these have been shown to be flawed in [147]. The distinguishers were based on non-existent keystream biases. Other than these, we do not find any attack reported on the RCR ciphers, although a number of attacks have been published on the other members (e.g., Py, TPypy, TPy6) of the Py family of ciphers. Furthermore, the use of rolling arrays and simple operations such as modular addition, bitwise XOR and circular shifts make the RCR ciphers remarkably efficient. In our preliminary analysis, we observed that the RCR-64 is one of the fastest stream ciphers available today. These reasons motivated us to revisit the RCR ciphers to perform a detailed security analysis and performance evaluation.

Contributions of this work. In [145], Sekar et al. did not make any specific recommendation on the key/IV size of the RCR ciphers, although the ciphers are as such able to support a wide range of key/IV sizes. In an attempt to further enhance the security of the RCR ciphers, we recommend using them with 256-bit keys and 384-bit IVs. We justify this recommendation in this Chapter.

In order to examine whether the RCR ciphers satisfy the expected statistical prop-

erties of an ideal cipher, we analysed their output sequences using the NIST statistical test suite [247]. We also performed the structural analysis of the RCR ciphers using six statistical tests proposed by Turan et al., which examine the key/IV setup and keystream generation algorithms of synchronous stream ciphers [248]. Besides, using the internal state diffusion test, which is proposed in this Chapter, we also verified whether the key and the IV correlate to the RCR ciphers' internal state. None of the tests could tell apart the RCR ciphers from a true random bit generator (TRBG).

Our security analysis suggests that the RCR ciphers are resistant to differential and linear cryptanalysis owing to their invertible IV setup algorithms and large secret S-boxes. Similarly, the ciphers are conjectured to be immune to algebraic cryptanalysis and cube attacks because of their large internal states which are thoroughly mixed during the key/IV setup. They preclude time-memory-data trade-off attacks with the recommended input sizes. Furthermore, the non-zero constant rotation that replaced the variable rotation used in the encryption phase of the TPy and the TPypy secure the RCR ciphers against classes of distinguishing attacks. We also present protected software implementations of RCR-64 and RCR-32 that are secure against (cache-)timing attacks and some prominent processor flag attacks.

In this Chapter, we also evaluate the performances of the RCR ciphers using the SUPERCOP tool kit developed by the eBACS project [155]. Our tests suggest that the protected implementations of RCR-64 and RCR-32 encrypt long messages with 256-bit keys on Intel Core-i5 680 processor at speeds of 2.56 and 4.06 cycles per byte, respectively. The protected implementation of the RCR–64 outperform some of the popular stream ciphers and block ciphers (in counter mode) like SIMON [212], SPECK [212], SNOW 2.0 [18], AES [30] implemented without the AES-NI instruction set, Sosemanuk [156] and HC-256 [23], in encrypting long messages. Consequently, it appears that the RCR ciphers are well suited for streaming applications. To the best of our knowledge, this is the first work to present a detailed study on the security and performance of the RCR ciphers.

Organisation of the Chapter. The remaining Chapter is organised as follows. In Sect. 6.2, we provide the specifications of the RCR ciphers. The results of our statistical analysis of the ciphers are discussed in Sect. 6.3. Our security analysis of the protected implementations of the RCR-64 and the RCR-32 are detailed in Sect. 6.4. We evaluate the performance of the RCR ciphers in Sect. 6.5 and conclude in Sect. 6.6.

6.2 Specifications of the Ciphers

6.2.1 RCR-64 and RCR-32

The internal states of the RCR ciphers are composed of a permutation P of 256 elements, 260-element array Y where each element is a 32-bit word and a 32-bit variable s. The RCR-64 and RCR-32 use identical key/IV setup which was originally designed for the ciphers TPy and TPypy (see Algorithms 16 and 17). Using two intermediate variables — a fixed permutation of 256 elements denoted by IP and a variable EIV whose size is equal to that of the IV — the key/IV setup mixes the secret key and the IV to generate the initial internal state. During an iteration of the keystream generation phase, the round functions of the RCR-64 and RCR-32 — denoted by $\mathcal{R}_1$ and $\mathcal{R}_2$, respectively — update the internal states and generate pseudorandom keystream words of size 64 bits and 32 bits, respectively (see Algorithm 18). While updating the internal state, the IV setup and the round function use the function $rotate(\cdot)$ to roll the arrays as $rotate(S) = \{S[1], S[2], \ldots, S[n-1], S[0]\}$, if S is the array $\{S[0], S[1], \ldots, S[n-2], S[n-1]\}$. Figure 6.1 provides a visual representation of the round functions of the RCR ciphers.

6.2.2 Recommended Key and IV Sizes

The key setup algorithms of the RCR ciphers, which have been adopted from the Py family of ciphers, are designed to use keys of size ranging from 1 to 256 bytes (in steps

Algorithm 16 Key setup

Require: A key, an IV and an initial permutation
Ensure: An array $Y[-3, -2, ..., 256]$

```
kb = size of the key k in bytes;
ivb = size of the IV in bytes;
L = -3; H = 256;

s = IP[kb - 1];
s = (s << 8) | IP[(s  ^ (ivb - 1)) & 0xFF];
s = (s << 8) | IP[(s  ^ k[0]) & 0xFF];
s = (s << 8) | IP[(s  ^ k[kb - 1]) & 0xFF];

for(j = 0; j < kb; j++)
{
        s = s + k[j]; s0 = IP[s & 0xFF];
        s = ROTL32(s, 8)  ^ (u32) s0;
}

for(j = 0; j < kb; j++)
{
        s = s + k[j]; s0 = IP[s & 0xFF];
        s ^= ROTL32(s, 8) + (u32) s0;
}

/*Initialise the array Y*/
for(i = L, j = 0; i <= H; i++)
{
        s = s + k[j]; s0 = IP[s & 0xFF];
        Y[i] = s = ROTL32(s, 8)  ^ (u32) s0;
        j = j + 1 (mod kb);
}
```

of a byte). Nevertheless, we recommend using 256-bit keys with the RCR ciphers due to the following two reasons. Firstly, according to the latest NIST recommendation, the security strength of a cryptographic algorithm, which is ideally equal to the key size in the case of a symmetric-key cipher, must be at least 112 bits [249]. Secondly, to provide equivalent security even against post-quantum adversaries, a rule of thumb is to double the key length due to Grovers' algorithm [250].

Similarly, the IV setup algorithms of the RCR ciphers are designed to use IVs of size ranging from 1 to 64 bytes (in steps of a byte). However, to prevent time-memory-data trade-off attacks, Dunkelman and Keller recommend that the IV size must be 1.5 times the key size [251]. Therefore, using a 384-bit IV with the RCR ciphers is recommended when the key size is 256 bits. Since the upper bound of the IV size of RCR ciphers

Algorithm 17 IV setup

Require: The Y and the IV

Ensure: The arrays $Y[-3,-2,...,256]$, $P[0,1,...,255]$ and the variable s

```
/*Create an initial permutation*/
ivb = size of IV in bytes;
L = -3;  H = 256;

v = IV[0] ^ ((Y[0] >> 16) & 0xFF);
d = (IV[1 (mod ivb)]  ^ ((Y[1] >> 16) & 0xFF)) | 1;

for(i = 0; i < 256; i++) { P[i] = IP[v]; v += d; }

/*Initialise s*/
s = ((u32) v << 24)  ^ ((u32) d << 16) ^ ((u32) P[254] << 8) ^ ((u32) P
    [255]);
s ^= Y[L] + Y[H];

/*Loop A: Mix the IV*/
for(i = 0; i < ivb; i++)
{
        s = s + IV[i] + Y[L + i];
        EIV[i] = s0 = P[s & 0xFF];
        s = ROTL32(s, 8) ^ (u32) s0;
}

/*Loop B: Update EIV*/
for(i = 0; i < ivb; i++)
{
        s += EIV[i + ivb - 1 (mod ivb)] + Y[H - i];
        s0 = P[s & 0xFF];
        EIV[i] += s0;
        s = ROTL32(s, 8) ^ (u32) s0;
}

/*Loop C: Update the rolling arrays and the variable s*/
for(i = 0; i < 260; i++)
{
        x0 = EIV[0] = EIV[0] ^ (s & 0xFF);
        rotate(EIV);
        swap(P[0], P[x0]);
        rotate(P);
        s = ROTL32(s, 8) + Y[H];
        Y[L] += s ^ Y[x0];
        rotate(Y);
}

s = s + Y[26] + Y[153] + Y[208];
if(s == 0) { s = (kb * 8) + ((ivb * 8) << 16) + 0x87654321; }
```

is 64 bytes as per the design, the key size cannot be more than 42 bytes, going by the recommendation of Dunkelman and Keller.

Algorithm 18 Round functions: $\mathcal{R}_1$ and $\mathcal{R}_2$

Require: $Y[-3, -2, ..., 256]$, $P[0, 1, ..., 255]$ and a 32-bit variable s
Ensure: A pseudorandom output Z

```
swap(P[0], P[Y[185] & 255]);
rotate(P);

s = s + Y[P[72]] - Y[P[239]];
s = ROTL32(s, 19);

/*Skip the next step for R2*/
Z = ((ROTL32(s, 25) ^ Y[256]) + Y[P[26]]);

Z = Z || ((s ^ Y[-1]) + Y[P[208]]);

Y[-3]=(ROTL32(s, 14) ^ Y[-3]) + Y[P[153]];
rotate(Y);
```

6.3 Statistical Analysis

One of the foremost objectives of an ideal stream cipher is to generate keystream bits that are distributed uniformly at random for a randomly chosen key. Therefore, the binary sequences generated using a secure stream cipher and a TRBG should be statistically alike. Similarly, the internal state of a stream cipher after the key/IV setup and the keystream generated from it should not be correlated to the key, the IV, or even to each other. The techniques used to examine whether RCR-64 and RCR-32 satisfy these statistical properties and their results are discussed in this Section. For all the tests, the ciphers were instantiated with 256-bit keys and 384-bit IVs.

6.3.1 Keystream analysis

The randomness properties of the output sequences generated by the RCR ciphers were analysed using the following tests available in the NIST statistical test suite: frequency, block frequency, runs, longest run, binary matrix rank, spectral, non-overlapping template matchings, overlapping template matchings, universal, linear complexity, serial, cumulative sums, approximate entropy, random excursions and random excursions variant [247].

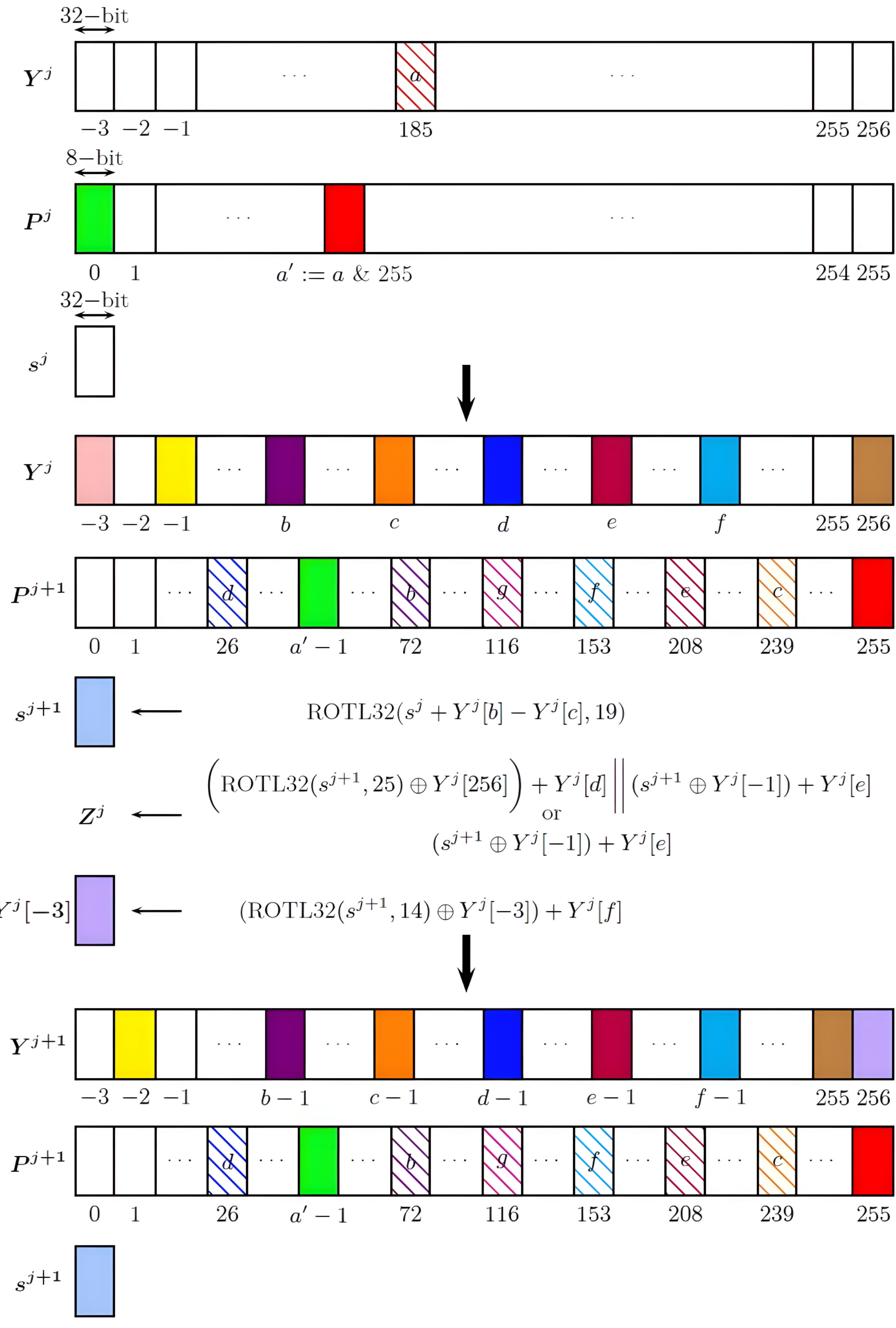


Figure 6.1: Visual representation of the round functions of the RCR ciphers, where (Y^j, P^j, s^j) and Z^j are the internal state at the beginning of jth round and the output word generated from it, respectively

Table 6.1: The results of the NIST statistical tests on the keystreams generated by the RCR ciphers, where p is the proportion of the sequences that passed the test

	RCR-64			RCR-32		
Statistical test	p	**P-value**	**Result**	p	**P-value**	**Result**
Frequency	0.986	0.258307	Pass	0.993	0.202268	Pass
Block frequency	0.989	0.883171	Pass	0.989	0.927677	Pass
Runs	0.992	0.858002	Pass	0.987	0.626709	Pass
Longest run	0.99	0.339271	Pass	0.987	0.538182	Pass
Binary matrix rank	0.991	0.279844	Pass	0.987	0.039329	Pass
Spectral	0.989	0.027313	Pass	0.987	0.587274	Pass
Overlapping template matchings	0.986	0.480771	Pass	0.984	0.133404	Pass
Universal	0.991	0.123755	Pass	0.989	0.809249	Pass
Linear complexity	0.992	0.930026	Pass	0.993	0.922855	Pass
Serial	0.989	0.279844	Pass	0.989	0.284024	Pass
Cumulative sums	0.983	0.092041	Pass	0.980	0.380407	Pass
Approximate entropy	0.992	0.532132	Pass	0.990	0.626709	Pass
Random excursions	0.985	0.026203	Pass	0.984	0.431143	Pass
Random excursions variant	0.980	0.012577	Pass	0.982	0.102526	Pass

Let us consider the hypothesis test given by the following.

- Null hypothesis: Output sequence of the cipher is random.
- Alternative hypothesis: Output sequence of the cipher is nonrandom.

For each cipher, 1000 keystream sequences of length 2^{20} bits were generated using randomly chosen key/IV pairs.[1] These binary sequences were given as input to the NIST test suite. According to the evaluation strategy of the test suite, the following conditions have to be satisfied by the cipher to pass each test:

1: At least 980 out of the 1000 keystream sequences have to pass the test because the proportion of sequences generated by a TRBG for which the test accepts H_0 has an expected value of 0.99 with a lower bound of 0.98 if the significance level is 0.01.

[1] The 640000-bit output of $AES_k(C)$ forms 1000 random key/IV pairs where AES_k is AES encryption in counter mode with a random 256-bit key k — generated by /dev/random — and C is $\{0\}^{640000}$.

2: The P-values calculated for the sequences must be uniformly distributed between 0 and 1. If we test the distribution of P-values using the chi-square goodness-of-fit test, with ten sub-intervals between 0 and 1, the P-value calculated for the resultant test statistic must be greater than 0.00001.

The results of the NIST statistical tests on the keystreams generated by the RCR-64 and RCR-32 are listed in Table 6.1. We can see that these tests could not tell apart the RCR ciphers from a TRBG.

6.3.2 Structural analysis

In [248], Turan et al. proposed six statistical tests for analysing synchronous stream ciphers. Among them, four tests — key-keystream correlation, IV-keystream correlation, frame correlation, and diffusion — intend to detect the correlations between the key/IV and a part of the keystream. The remaining two tests — IV-internal-state correlation and internal-state-keystream correlation — are used to evaluate input/output correlations with the internal state. In addition to these tests, we propose the internal state diffusion test which verifies whether the key and the IV are correlated to the internal state. Using these tests, we performed our structural analysis of RCR ciphers.

Key-keystream correlation test. The correlation between the key and the first 256 bits of the keystream was evaluated with this test. For some fixed IV, 2^{20} keystreams, each 256 bits long, were generated using 2^{20} keys chosen uniformly at random. Each key was XORed with its corresponding keystream, and the Hamming weights of these XOR sums were computed. For an ideal cipher, the Hamming weight should follow the binomial distribution with $n = 256$ and $p = 0.5$.

IV-keystream correlation test. This test was used to evaluate the correlation between the IV and the first 384 bits of the keystream. For some fixed key, 2^{20} keystreams, each of length 384 bits, were generated using 2^{20} IVs chosen uniformly at random.

The keystreams were XORed with the respective IVs, and the Hamming weights of these XOR sums were computed. Similar to the key-keystream correlation test, the Hamming weights should follow the binomial distribution with $n = 384$ and $p = 0.5$ for the ciphers to be ideal.

Frame correlation test. The correlation between the keystream frames generated using similar IVs was examined using this test. Let $iv_j := r \parallel j$ for $j = 0, 1, \dots, 2^{20} - 1$, where r and j are the fixed 364 bits and variable 20 bits of the IV, respectively, form the set of similar IVs.[2] For some fixed key and r, 2^{20} keystreams (each of length 256 bits) were generated using the similar IVs. A binary matrix of size $2^{20} \times 256$ was created whose jth row contained the keystream bits generated using iv_j. The column weights of the matrix, which were computed by counting the number of ones in each column, follow the binomial distribution with $n = 2^{20}$ and $p = 0.5$ in the ideal case. Since n is large, the binomial distribution can be approximated to the normal distribution with mean 2^{19} and variance 2^{18}.

Diffusion test. Using this test, we examined the diffusion property of each bit of the key and the IV on the keystream. A 256-bit reference keystream was generated using a randomly chosen reference key/IV pair of length 640 bits. To analyse the effect of each bit of the key/IV pair on the keystream, difference vectors were generated by XORing the reference keystream with each keystream generated using the key/IV pairs that differ from the reference key/IV pair by a single bit. A binary matrix of size 640×256 was created using these difference vectors. This process was repeated 2^{10} times, and the generated matrices were added in real numbers. For an ideal cipher, each element of the resultant matrix, which is the sum of 2^{10} random bits, is expected to follow the binomial distribution with $n = 2^{10}$ and $p = 0.5$.

[2] There are 2^{20} similar IVs for an arbitrary 364-bit r.

IV-internal-state correlation test. This test, which is very similar to the frame correlation test, helps in analysing the effect of similar IVs on the internal state of the cipher. For some fixed key, 2^{20} internal states were generated using the similar IVs, as done in frame correlation test, and from them, a binary matrix of size $2^{20} \times 10400$ was created. The column weights of the matrix have to ideally follow the normal distribution with mean 2^{19} and variance 2^{18}.

Internal-state-keystream correlation test. For an ideal cipher, the Hamming weight of the internal state should not be correlated to the Hamming weight of the keystream. This feature was examined using the internal-state-keystream correlation test. We know that two components of the internal state of the RCR ciphers, namely Y and s, are distributed uniformly at random, and the third component P is a random permutation. Without using the key/IV setup, the internal state was initialised in such a way that P was a random permutation, and Y and s contained random data with low (lesser than 4000) or high (greater than 6000) Hamming weights. From 2^{10} such internal states, keystream sequences of length 10400 bits were generated. In case of an ideal cipher, the Hamming weights of these keystreams must follow the binomial distribution with $n = 10400$ and $p = 0.5$. Since n is large, the binomial distribution can be approximated to the normal distribution with mean 5200 and variance 2600.

Internal state diffusion test. Each bit of the key and the IV of an ideal cipher should affect its entire internal state. To analyse this feature, 2^{10} reference internal states were generated using 2^{10} randomly chosen reference key/IV pairs (each of length 640 bits). A difference vector can be generated by XORing the reference internal state with the internal state generated from a key/IV pair that differs from the reference key/IV pair by a single bit. For each reference key/IV pair, 640 difference vectors were generated. As in the case of diffusion test, 2^{10} binary matrices, each of size 640×256, created from 2^{10} reference key/IV pairs were added in real numbers.

Table 6.2: The mean P-value computed for each structural analysis test on RCR-64 and RCR-32

Test	RCR-64	RCR-32
Key-keystream correlation	0.50636	0.46986
IV-keystream correlation	0.51372	0.51380
Frame correlation	0.45901	0.49905
Diffusion	0.35263	0.34547
IV-internal-state correlation	0.46097	0.48847
Internal-state-keystream correlation	0.43934	0.43854
Internal state diffusion	0.43759	0.46512

Each element of the resultant matrix, which is the sum of 2^{10} random bits, are expected to follow the binomial distribution with $n = 2^{10}$ and $p = 0.5$ if the cipher is ideal.

The RCR-64 and RCR-32 were subjected to each of these correlation tests 100 times, and the chi-square goodness-of-fit test was used to verify if the observed data followed the expected distributions. Let us consider the hypothesis test given by the following.

- Null hypothesis: The observed data follow the expected distribution.
- Alternative hypothesis: The observed data do not follow the expected distribution.

The mean P-value computed for each structural analysis test on RCR-64 and RCR-32 is listed in Table 6.2. The value must be greater than or equal to 0.01 to conclude that the observed data follow the expected distribution. From the results, it is clear that the key/IV, the internal state and the keystream of the RCR ciphers are uncorrelated.

6.4 Security Analysis

6.4.1 Resistance to differential cryptanalysis

Differential cryptanalysis of synchronous stream ciphers often concern the effect of differences in the IVs on the internal state or the keystream. In [242], Wu and Preneel found that a pair of IVs with a certain difference can initialise the stream cipher Py [25]

to identical internal states. The same applies to Pypy [25, 152]. Using such pairs of IVs, key recovery attacks can be built on Py and Pypy [242, 243]. The differential attacks originated due to an additional mixing loop in the IV setup to mix the IV into the internal state. The additional loop was removed when Biham and Seberry tweaked the weak IV setup of the Py and Pypy [151]. The designers also ensured that the tweaked IV setup is invertible to prevent any collisions in the internal state [151]. Sekar et al. adopted the modified IV setup for the RCR ciphers. Our investigations indicate that the modified IV setup resists differential attacks of a similar nature as those of [242] and [243].

We see from Algorithm 17 that any difference introduced in the IV propagates immediately to s through the loops A and B. Later, during each iteration of loop C, the differences in s affect P and Y. Figure 6.2 illustrates the propagation of the XOR difference through the loops A and B.[3] The permutation P, which is a randomly chosen 8×8-bit S-box, make it challenging to model the differential propagation. The expected value of the highest probability of a (non-trivial) differential characteristic is at the most $m/2^{m-1}$ for a randomly chosen $m \times m$-bit bijective mapping [252]. Therefore the expected maximum probability of the differential characteristic for P is 2^{-4}. Similarly, the XOR differential probability of the modular addition is upper bounded by $2^{-\beta}$ if the number of bit positions, excluding the most significant bit, at which the input/output differences of the modular addition differ equals β [253]. For an ivb-byte IV, the XOR difference introduced at the $(j+1)$th byte position propagates through P for $ivb - j$ and ivb times in loop A and B, respectively. Similarly, there are ivb active modular additions affecting the differential propagation through loop B. Therefore, if the IV size is 48 bytes, the upper bound of the differential probability across the loops A and B is given by $2^{-4(48-j)-48(\beta+4)}$, where $0 \leq j \leq 47$ and $0 \leq \beta \leq 31$. We can see that the probability of the best differential characteristic across the loops A and B is upper bounded by 2^{-244}, under the assumption that β equals one. The propagation of the differences through Y, which acts as an 8×32-bit S-box, further lowers the differential probability, making it

[3] To simplify the analysis, the IV mixing operation of the loop A of Algorithm 17 is assumed to be linear.

seemingly impossible to build differential attacks faster than exhaustive key search.

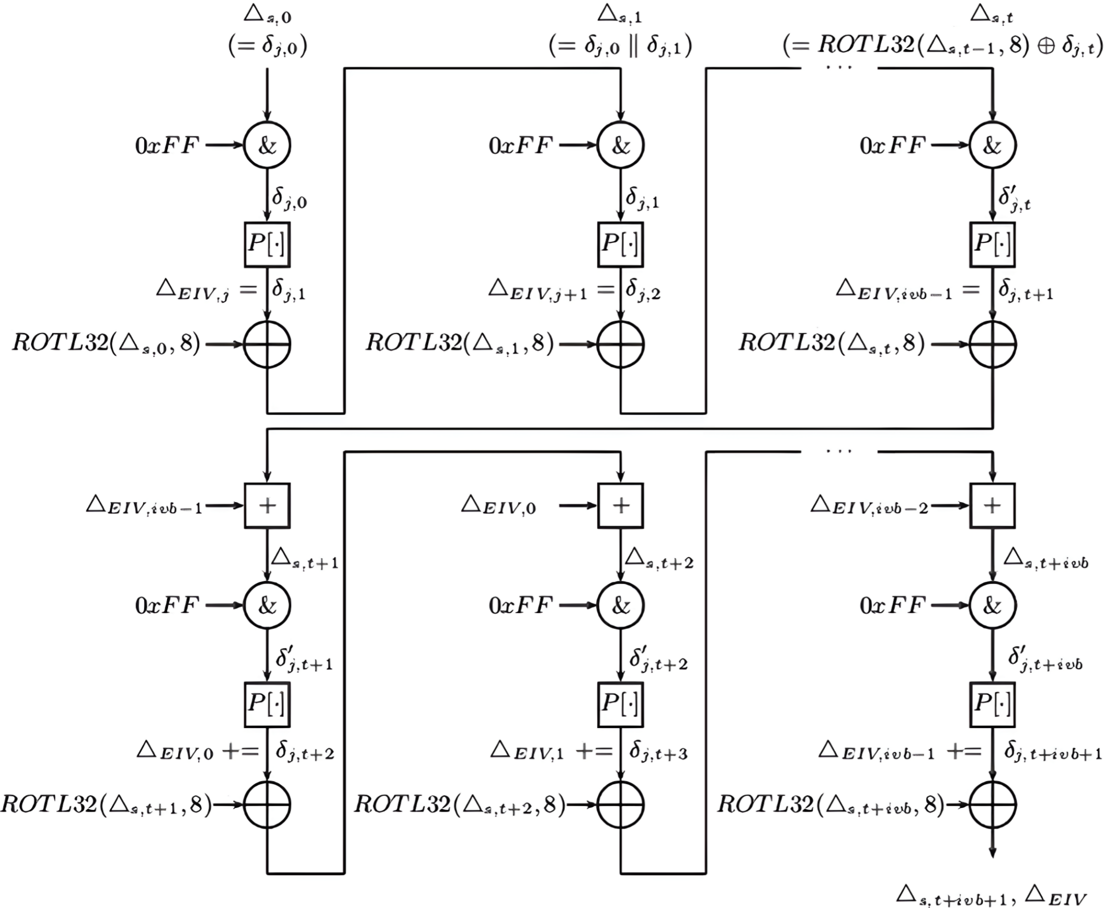


Figure 6.2: Propagation of the XOR difference $\triangle_{s,0} := \delta_{j,0}$ through the loops A and B of Algorithm 17 to $(\triangle_{s,t+ivb+1}, \triangle_{EIV})$, where $\delta_{j,0}$ is the difference at the $(j+1)$th byte of the IV, ivb is the IV size in bytes, $\triangle_{s,i+1}$ and $\delta_{j,i+1}$ are the differences in s and the output of $P[\cdot]$, respectively, after the $(i+1)$th step ($0 \leq i \leq t + ivb + 1$; $t = ivb - 1 - j$) and, $\triangle_{EIV}$ and $\triangle_{EIV,i}$ are the differences in EIV and its $(i+1)$th byte ($0 \leq i \leq ivb - 1$), respectively

6.4.2 Resistance to linear cryptanalysis

The linear cryptanalysis of stream ciphers, devised by Golić [116], exploits the existence of linear approximations between some keystream bits and the internal state bits or the key/IV bits to recover the secret internal state or the key. Even if it is not possible to recover the internal state or the key, linear cryptanalysis might be useful in finding good distinguishers against the ciphers. The keystream generation processes of the RCR

ciphers involve 8×8-bit and 8×32-bit key-dependent secret S-boxes, namely P and Y, respectively. The ciphers Blowfish [38] and HC-256 [23], which use 8×32-bit and 32×32-bit secret S-boxes, respectively, are resistant to linear cryptanalysis due to their large secret S-boxes. Moreover, since P and Y are updated during each round of the keystream generation process, it will be difficult to find linear relations linking the input and output bits of the S-boxes. Due to these reasons, we conjecture that the RCR ciphers are secure against linear cryptanalysis.

6.4.3 Resistance to algebraic cryptanalysis

Several stream ciphers, whose operations can be represented as a system of multi-variate algebraic equations, have been found to be vulnerable to algebraic attacks. The adversary solves the system of equations to recover the secret key or the internal state. It is powerful against ciphers with simple algebraic structures such as functions of low algebraic degrees. Algebraic attacks are therefore common against LFSR-based stream ciphers with small internal states such as LILI-128, E0, Toyocrypt, SOBER-t32, SOBER-t16, SSG and Trivium [103, 105, 254, 255, 256]. Whereas, array-based ciphers like RC4 [21], RC4A [257], $RC4^+$ [258], VMPC [259], IA, IBAA, ISAAC [260], NGG [261], GGHN [262], Spritz [22], HC family [23, 24] and Py family [152, 25, 151], which have large internal states are not vulnerable to algebraic cryptanalysis, to the best of our knowledge. Hence, being array-based ciphers with large internal states of size 10,400 bits each, it is difficult to construct algebraic attacks on the RCR ciphers too.

6.4.4 Resistance to cube attacks

The cube attack is a cryptanalytic technique, used against symmetric-key cryptosystems, proposed by Dinur and Shamir [263]. Let $k := (k_1, k_2, \ldots, k_n)$ and $v := (v_1, v_2, \ldots, v_m)$ be an $n-$bit secret key and an $m-$bit IV, respectively, and $f(k, v)$ be a polynomial function over GF(2) that generates the first keystream bit of a stream cipher instantiated with k and v. Let C_I — referred to as a *cube* — be the set of $2^{|I|}$

values of v such that the variables $\{v_{i_1}, v_{i_2}, \ldots, v_{i_{|I|}}\}$ take all possible combinations of binary values and the remaining variables are fixed to some arbitrary binary values, where $I = \{i_1, i_2, \ldots, i_{|I|}\} \subset \{1, 2, \ldots, m\}$. If $f(k, v)$ can be decomposed as:

$$f(k, v) = t_I \cdot p(k, v) + q(k, v)\,,$$

where $t_I = v_{i_1} \cdot v_{i_2} \cdot \ldots \cdot v_{i_{|I|}}$, $p(k, v)$ — known as a *superpoly* — is a polynomial independent of $\{v_{i_1}, v_{i_2}, \ldots, v_{i_{|I|}}\}$ and $q(k, v)$ misses at least one variable from $\{v_{i_1}, v_{i_2}, \ldots, v_{i_{|I|}}\}$, then we get:

$$\begin{aligned} \bigoplus_{C_I} f(k, v) &= \bigoplus_{C_I} t_I \cdot p(k, v) + \bigoplus_{C_I} q(k, v) \\ &= p(k, v)\,. \end{aligned} \tag{6.1}$$

The core idea of the cube attack is to detect some superpolies and exploit them to recover k. The attack works in the following two phases: preprocessing and online. In the preprocessing phase, the attacker searches for the superpolies $p(k, v)$ and in the online phase, the set of equations generated using (6.1) are solved to recover the secret key. The success of a cube attack depends on the detection of a superpoly whose algebraic degree is low enough to enable efficient recovery of the key bits. The algebraic degrees of the polynomials representing the RCR ciphers will be too large considering the extensive mixing of their internal state variables during the key/IV setup through multiple rounds of nonlinear operations. As it will be computationally hard to detect low degree superpolies in such cases, the RCR ciphers must be resistant to cube attacks.

6.4.5 Resistance to time-memory-data trade-off (TMDTO) attacks

TMDTO attack is a generic technique applicable to both block ciphers and stream ciphers. It inverts a one way function f at a single point in the range of the function using the following two phases:

1: A preprocessing phase during which the attacker constructs several tables which store a set of starting points, chosen from the range of f, and their corresponding ending points, generated by recursively applying f or its variants for a fixed number of rounds.

2: An online phase during which the attacker searches the precomputed tables to recover the unknown input to f that generated the real-time data.

The five parameters that define a TMDTO attack are the size of the search space N, the time required by the preprocessing phase R, the amount of memory required to store the tables M, the time required by the online phase T and the amount of real-time data D.

The objective of the first TMDTO attack on stream ciphers, independently proposed by Babbage [264] and Golic [265], was to invert the function that maps the internal state of the cipher of size $\log_2(N)$ bits to $\log_2(N)$ keystream bits. The D possible windows of $\log_2(N)$ consecutive bits taken from $D + \log_2(N) - 1$ keystream bits constituted the real-time data needed for this attack. The trade-off curve of this attack was defined by $TM = N$ and $R = M$ for any $1 \leq T \leq D$. This attack was later improved by Biryukov and Shamir by reducing the number of matrices by a factor of D [266]. This resulted in a better trade-off curve $TM^2D^2 = N^2$ for $R = N/D$ and $D^2 \leq T \leq N$. Since the internal states of the RCR ciphers are of size 10,400 bits, neither of these TMDTO attacks will be better than the exhaustive key search.

Another type of TMDTO attack on stream ciphers was proposed by Hong and Sarkar [267], which focussed on inverting the function that maps the key/IV pair to keystream bits of size $\log_2(\kappa) + \log_2(\nu)$ bits, where $\log_2(\kappa)$ and $\log_2(\nu)$ are the sizes of the key and the IV in bits, respectively. Being similar to Biryukov and Shamir's attack and having a search space constituted by the key space and IV space, Hong and Sarkar's attack has the trade-off curve $TM^2D^2 = \kappa^2\nu^2$, for $T \geq D^2$. In order to exploit the fact that the IV is publicly known, Dunkelman and Keller proposed a new approach to Hong and Sarkar's attack by defining a slightly different function f which maps the key to keystream bits,

for a fixed IV [251]. During the preprocessing phase of this attack, ν/D IVs are chosen and using the function corresponding to each IV, a set of tables is generated. When a keystream is available for one of the chosen IVs, the attacker applies the online phase of the TMDTO attack using the tables prepared for that IV. Similar to the previous attack, this one also has the trade-off curve $TM^2D^2 = \kappa^2\nu^2$. For a stream cipher to achieve n-bit security against TMDTO attacks, Hong and Sarkar, and Dunkelman and Keller recommended using IVs of size n and $1.5n$ bits, respectively, where n is the key size in bits. As the RCR ciphers follow these recommendations, they are resistant to the TMDTO attacks due to Hong and Sarkar, and Dunkelman and Keller.

6.4.6 Resistance to distinguishing attacks

Distinguishing attack enables an attacker to tell apart a cipher from an ideal cipher who exploits the biases in its output to build the distinguisher. The predecessors of the RCR ciphers, namely the TPy and TPypy [151], are vulnerable to distinguishing attacks due to the weaknesses in their KGAs [240, 241, 246, 245, 268]. The rotation of s in the KGAs, given by $\texttt{ROTL32}\big(s, ((P[116]+18) \,\&\, 31)\big)$, was targeted by these attacks, and the distinguishers depend on the simultaneous occurrence of some of the following events:

$$P^{r_1}[116] \equiv -18 \pmod{32}\,, \text{ for some round } r_1\,, \tag{6.2}$$

$$P^{r_2}[116] \equiv 7 \pmod{32}\,, \text{ for some round } r_2\,, \tag{6.3}$$

$$P^{r_3}[116] \equiv 0 \pmod{32}\,, \text{ for some round } r_3\,, \tag{6.4}$$

$$P^{r_4}[116] \equiv -4 \pmod{32}\,, \text{ for some round } r_4\,, \tag{6.5}$$

$$P^{r_5}[116] \equiv 3 \pmod{32}\,, \text{ for some round } r_5\,, \tag{6.6}$$

where P^r denotes the P at the beginning of rth round of encryption. The events (6.2)-(6.6) translate to the cyclic left rotation of s by 0, 25, 18, 22 and 21 bits, respectively. Since the RCR ciphers use a constant rotation given by $\texttt{ROTL32}(s, 19)$, they preclude distinguishing attacks of the kind mentioned above. The related-key distinguishing attacks

on the ciphers TPy and TPypy by Sekar et al. [145] are also precluded thus.

In [146], Ding et al. proposed related-key distinguishers on the Py family of stream ciphers, including the RCR-64 and RCR-32. These appear to be the only published attacks on the RCR ciphers. The distinguishers depend on three events, the simultaneous occurrence of which gives: $\widehat{Z} := Z^1_{1(0)} \oplus Z^2_{1(0)} \oplus Z^1_{2(0)} \oplus Z^2_{2(0)} = 0$, where Z^j_i is the second output word of the jth round of encryption using the key k_i (for i = 1, 2). If any of the events does not occur, Ding et al. incorrectly assume that Pr$(\widehat{Z} = 0) = 0.5$. Consequently, they detect a bias in $\widehat{Z}$, leading to related-key distinguishing attacks on RCR–64 and RCR–32. The flawed computations by Ding et al. and the non-existence of their keystream bias are established in [147].

Let (P^r, Y^r, s^r) denotes the internal state at the beginning of rth round of encryption. All the distinguishing attacks on the Py family of ciphers, including the related-key attacks presented in [145], exploited the following weakness due to the variable rotation of s: when certain conditions on the elements of the array P are satisfied then $s^r_{(i)}$ equals $s^{r+2}_{(j)}$, where $0 \leq i, j \leq 31$. If the variable rotation is replaced with a constant rotation, which rotates s by some non-zero factor (say c), we get:

$$s^r = \texttt{ROTL32}\left(s^{r-1} + Y^{r-1}[P^r[72]] - Y^{r-1}[P^r[239]], c\right). \tag{6.7}$$

If $k = i - c$ (mod 32), we get:

$$s^r_{(i)} = s^{r-1}_{(k)} \oplus Y^{r-1}[P^r[72]]_{(k)} \oplus (Y^{r-1}[P^r[239]]_{(k)})' \oplus \epsilon^r_{(k)},$$

where ϵ^r is the carry term generated in (6.7), and $\epsilon^r_{(0)} = 1$. Similarly, we have:

$$s^{r+2}_{(j)} = s^{r+1}_{(l)} \oplus Y^{r+1}[P^{r+2}[72]]_{(l)} \oplus (Y^{r+1}[P^{r+2}[239]]_{(l)})' \oplus \epsilon^{r+2}_{(l)}, \tag{6.8}$$

$$s^{r+1}_{(l)} = s^{r}_{(m)} \oplus Y^{r}[P^{r+1}[72]]_{(m)} \oplus (Y^{r}[P^{r+1}[239]]_{(m)})' \oplus \epsilon^{r+1}_{(m)}, \tag{6.9}$$

$$s^{r}_{(m)} = s^{r-1}_{(n)} \oplus Y^{r-1}[P^{r}[72]]_{(n)} \oplus (Y^{r-1}[P^{r}[239]]_{(n)})' \oplus \epsilon^{r}_{(n)}, \tag{6.10}$$

where l, m and n denote $j-c \pmod{32}$, $l-c \pmod{32}$ and $m-c \pmod{32}$, respectively. The expression for $s^{r}_{(i)} \oplus s^{r+2}_{(j)}$ that is obtained by substituting (6.9) and (6.10) in (6.8) contains the term $s^{r-1}_{(k)} \oplus s^{r-1}_{(n)}$ which takes the value 0 with uniform probability of 0.5 when $n \neq k$. Since $k = i - c \pmod{32}$, $n = j - 3c \pmod{32}$ and $c \neq 0$, n will not be equal to k when $i = j$. Likewise, if $i \neq j$, the remaining Y-terms in the expression do not cancel out, under the assumption that they are uncorrelated. Therefore, by using any non-zero constant such as $c = 19$, the problem behind a multitude of distinguishing attacks on the Py family of ciphers is fixed. It is therefore conjectured that the RCR ciphers are immune to distinguishing attacks that are faster than exhaustive key search.

6.4.7 Resistance to (cache-)timing attacks

To reduce the time required to access data from the main memory, microprocessors use a fast and small memory known as the cache memory. It stores copies of the data from frequently accessed locations of the main memory. A memory access operation will be faster when the data is available in the cache than when it is unavailable. A (cache-)timing attack exploits this time difference in memory access to recover the secret internal state of a cipher. As proposed by Osvik et al., Evict + Time and Prime + Probe are the two methods to implement a (cache-)timing attack [137]. In Evict + Time method, the adversary will evict a set of chosen cache blocks and measure the time taken for encryption. If the time taken is more when compared to the encryption without evicting the cache blocks, she will confirm that the chosen cache blocks have been accessed during encryption. In the Prime + Probe method, the adversary will preload the cache before encryption with known data and the time taken to access it after encryption is measured. The time taken to access a block of data will be more if the corresponding cache block has been evicted during encryption.

In general, (cache-)timing attacks are built on unprotected implementations of ciphers to recover the contents of the secret state, which act as indices to lookup tables or arrays. Being array-based constructions, RCR ciphers will be vulnerable to cache at-

tacks if not implemented securely. In order to protect against Evict + Time and Prime + Probe methods of (cache-)timing attack, Osvik et al. recommend normalising the cache state just before and just after the encryption by loading all the arrays into the cache [137]. To achieve the aforementioned normalisation of the cache state in the reference implementations of RCR–64 and RCR–32, the internal state is copied to temporary arrays just before the encryption, and the updated state is copied back from the temporary arrays after the encryption. Nevertheless, this technique provides little protection if and when the adversary is able to access the cache memory during encryption.

6.4.8 Resistance to carry flag attacks

In [129], Kelsey et al. introduced processor flag cryptanalysis, that exploits the carry flag or overflow flag bits of the CPU status register. Unprotected implementations of symmetric-key ciphers such as RC5, Idea, Twofish and SPECK, the hash function Streebog and some public-key cryptosystems, which use addition to operate on their secret states, have been found to be vulnerable to carry flag attacks [129, 138, 131, 133, 132]. As conjectured in [138], the overflow flag might also be helpful in some instances to augment the carry flag attacks as it indicates the carry at the second most significant bit position. Similarly, the parity flag available in certain microprocessors, which indicates if the number of ones in the binary representation of the result of an instruction is odd or even, might also be useful in reducing the complexity of these attacks.

Since the final keystream generation step in the round functions of the RCR ciphers, given by $Z = ((s \oplus Y[-1]) + Y[P[208]])$, involves the addition of two 32-bit variables derived from the secret state, unprotected implementations of these ciphers might be vulnerable to carry flag attacks. Therefore, it is necessary to mask the carry, overflow and parity flag bits soon after the encryption in order to prevent the adversary from extracting any meaningful information from them. In Chapter 4, we have discussed two types of adversaries in carry flag cryptanalysis. The first type can detect the carry, overflow and parity flags at the end of encryption. Under this adversarial assumption,

the attack can be prevented by (re)setting the flags just after encryption. The second type of adversary, who can detect the bit-flips in the flags, will overcome this protection as the final status of the flags is known. In order to prevent such adversaries, (re)setting the flags after encryption should be done uniformly at random.

Therefore, to protect the reference implementation of RCR ciphers from carry flag attacks, we suggest the following masking operation immediately following the key/IV setup and encryption:

$$r = r_1 + \left(r_2 \parallel 1 \parallel \{0\}^{30}\right), \tag{6.11}$$

where r_1 and r_2 respectively are 32-bit and 1-bit numbers that are uniformly distributed at random. Since the addition of $1 \parallel \{0\}^{30}$ with a 31-bit uniformly distributed random number generates an outgoing carry at the 31st bit position with probability 0.5, the overflow flag will be set randomly in (6.11). Similarly, the 1-bit addition $c_{(31)} + r_{1,(31)} + r_2$, where $c_{(31)}$ is the overflow flag, sets the the carry flag uniformly at random. With the output r being uniformly distributed at random the parity flag will also be set with probability 0.5.

Since it is feasible to distinguish between $0 \rightarrow 1$ and $1 \rightarrow 0$ bit transitions in certain implementations [218, 219], we can consider a third type of adversary who can detect the direction of the bit transitions in the flags under consideration. She will not be able to detect the flags if the flag bits do not flip after the masking operation.

6.5 Performance Evaluation

We evaluated the performances of the following two implementations of each RCR cipher in software: the unprotected implementation, which is vulnerable to carry flag attacks, and the protected implementation, which masks the processor flags as explained in Sect. 6.4.8. The software performances of these implementations to encrypt messages with 256-bit keys on Intel Core-i5 680 processor were measured using the SUPERCOP

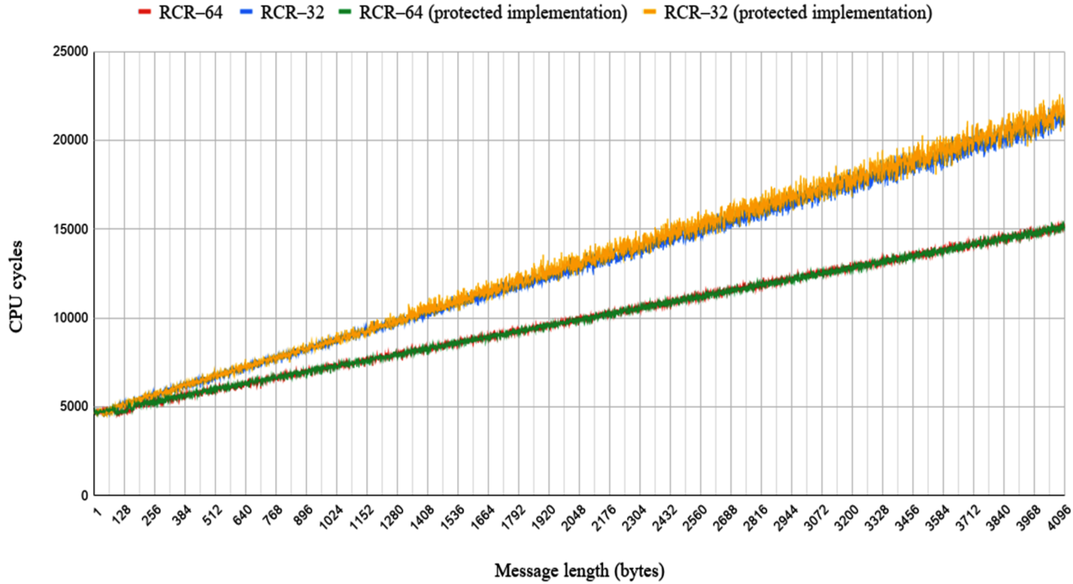


Figure 6.3: Encryption timings (in CPU cylces) of unprotected and protected implementations of the RCR ciphers for all message lengths between 1 and 4096 bytes

tool kit developed by the eBACS project [155].[4] The number of CPU cycles required to encrypt messages of lengths ranging between 1 and 4096 bytes by the tested implementations is plotted in Figure 6.3. Since the number of clock cycles measured by the tool includes the time taken for key/IV setup, it is evident from Figure 6.3 that the key/IV setup of the RCR ciphers consumed nearly 5000 clock cycles on the tested platform. The test results also prove that the extra code included in the protected implementation to prevent processor flag attacks has only a marginal effect on the performance.

We compared the performance of the RCR ciphers with various 256-bit stream cipher implementations available in the SUPERCOP tool kit, including the block ciphers AES [30], SIMON [212] and SPECK [212] in counter mode. The performance in cycles per byte to encrypt 64-byte, 576-bye, 1536-byte, 4096-byte and much longer messages on Intel Core-i5 680 processor by these stream ciphers are listed in Table 6.3. It can be seen that the protected and unprotected implementations of the RCR–64 outperform most of the popular ciphers like SIMON, SPECK and SNOW 2.0 [18] (ISO/IEC standards [270, 143, 19]), AES implemented without the AES-NI instruction set [158], Sosemanuk

[4]The procedure to evaluate the performance of RCR ciphers using the SUPERCOP toolkit is available at [269].

[156] (an eSTREAM final portfolio cipher [157]) and HC-256 [23] (the 256-bit variant of the eSTREAM final portfolio cipher HC-128 [157]), in encrypting long messages. It must be noted that the reduced performance while encrypting shorter messages is due to the computationally heavy key/IV setup. The performance evaluation also proves that the RCR–64 and RCR–32 are faster than their predecessors TPy and TPypy [151], respectively, due to the absence of three operations — an array access, an addition and a bitwise AND — every encryption round.

Table 6.3: The performances of a few stream ciphers, including the AES, SPECK and SIMON in counter mode, to encrypt 64-byte, 576-byte, 1536-byte, 4096-byte and much longer messages with 256-bit keys on Intel Core i5-680 processor measured using SUPERCOP toolkit

Cipher	**Performance in cycles per byte to encrypt**				
	64-byte message	**576-byte message**	**1536-byte message**	**4096-byte message**	**longer message**
ChaCha8 [85]	3.50	1.51	1.30	1.27	1.24
ChaCha12 [85]	4.59	2.07	1.80	1.77	1.75
AES CTR[a] [30]	7.39	2.81	2.42	2.34	2.30
RCR–64[b]	73.97	10.30	5.62	3.68	2.52
Salsa20/12 [86]	5.33	2.87	2.60	2.55	2.53
RCR–64[c]	75.27	10.70	5.65	3.67	2.56
ChaCha20 [85]	6.89	3.20	2.77	2.74	2.72
TPy [151]	113.03	15.14	7.53	4.64	2.93
TPy6 [151]	51.14	8.53	5.10	3.79	2.96
CryptMT v3 [159]	13.61	5.09	4.96	3.46	3.05
SPECK128/256 CTR [212]	6.81	3.72	3.41	3.33	3.28
Salsa20/8 [86]	4.17	2.15	1.94	3.04	3.40
Salsa20/20 [86]	7.97	4.32	3.92	3.88	3.85
RCR–32[b]	71.61	12.13	6.96	5.10	3.92
SNOW 2.0 [18]	15.45	5.22	4.41	4.11	3.93
RCR–32[c]	75.19	11.93	7.11	5.30	4.06
HC-256 [23]	754.89	87.51	35.44	15.89	4.18
Sosemanuk [156]	22.67	6.44	5.00	4.45	4.19
TPypy [151]	113.22	16.35	8.87	5.89	4.21
SIMON128/256 CTR [212]	20.44	15.61	9.12	7.71	6.83
AES CTR [30]	27.31	18.86	18.18	17.96	17.82

[a]Implemented with AES-NI instruction
[b]Unprotected implementation
[c]Protected implementation

6.6 Conclusions

In this Chapter, we have revisited the software-efficient stream ciphers RCR-64 and RCR-32, designed by Sekar et al. as strengthened variants of the ciphers TPy and TPypy, respectively. To the best of our knowledge, there are no prior published results analysing the security and performance of the RCR ciphers in detail. To further improve the security, we recommend that the RCR ciphers be used with 256-bit keys and 384-bit IVs. In addition to the fact that the RCR ciphers have remained unbroken since they were published in 2007, our thorough analysis suggests that the ciphers, especially with the recommended input sizes, are secure against differential, linear, algebraic, cube, time-memory-data trade-off and distinguishing attacks. We have also proposed protected software implementations of the ciphers which are secure against (cache-)timing attacks and some processor flag attacks. Our performance analysis suggests that the protected implementation of the RCR-64 encrypts long messages at speeds comparable to some of the fastest stream ciphers available today. In conclusion, therefore, the ciphers appear to be well suited for widespread deployment in software applications.

Chapter 7

Closing Remarks

7.1 Conclusions

In this thesis, we have dealt with the the security analysis of some of the popular symmetric-key algorithms, including lightweight constructions, and their unprotected implementations. To begin with, we discussed the importance of probability assumptions and computations in symmetric-key cryptanalysis using a case study. Then we presented cryptanalytic results on two families of lightweight stream ciphers covered by a US patent and unprotected implementations of a family of lightweight block ciphers standardised by ISO/IEC for RFID devices. Next, we presented side-channel attacks on a family of MAC algorithms defined in the Russian cryptographic standards. Finally, we proposed ways to protect the software implementations of a family of stream ciphers with detailed security and performance evaluations.

The first contribution of this thesis highlights some potential flaws in probability-based cryptanalysis. As a case study, we reviewed the related-key distinguishing attacks on the stream ciphers Py, Pypy, TPy, TPypy, RCR-64 and RCR-32 proposed in a paper published in the *Journal of Universal Computer Science* in detail. We could establish the flawed computations that led to the alleged attacks, and the non-existence of the keystream biases detected in the Py family of stream ciphers.

The second contribution is on the algorithmic cryptanalysis side, where we presented distinguishing attacks on the Welch-Gong (WG) family of patented (ultra-)lightweight stream ciphers. These ciphers were proposed to secure lightweight applications like RFID systems and 4G/5G networks. Our attacks exploited the input-output correlations in the nonlinear transformations used by these ciphers. Though our attacks on the lightweight members of WG are certificational in nature, the attacks on the ultra-lightweight members are highly practical, requiring fewer than $2^{29.07}$ keystream samples for nearly guaranteed success in distinguishing the ciphers from random. Due to the low attack complexities, we could also experimentally verify our attacks on ultra-lightweight WG ciphers. To the best of our knowledge, these are the first attacks on these ciphers.

We have also contributed on the side-channel analysis side by presenting carry flag attacks on unprotected implementations of SPECK family of lightweight block ciphers and HMAC–Streebog family of MAC algorithms. SPECK, an ISO/IEC standard for RFID devices, and HMAC–Streebog, a Russian cryptographic standard, use modular addition making them vulnerable to carry flag attacks. This is a fairly well-known technique originally proposed by Kelsey et al. [129] and thoroughly explored in a few papers [131, 132, 133, 134]. Nevertheless, to the best of our knowledge, this thesis presents the first results analysing the resistance of unprotected implementations of SPECK and HMAC-Streebog to carry flag cryptanalysis.

Being an ISO standard and designed by the researchers of the NSA, SPECK is a potential candidate for wide-scale deployment in lightweight applications. Our attacks, which work on the full SPECK, are comparatively more feasible than the other attacks applicable on the full ciphers, which require fault injections, due to the weaker assumptions we make. The thesis has also discussed the details of the vulnerable implementations of SPECK ciphers and proposed a countermeasure to preclude our attacks.

We presented two types of side-channel attacks on HMAC-Streebog: passive and active. Our passive attacks, which assume that the inner hash of the HMAC and the carry flag at the end of MAC generation are known to the attacker, can recover one

bit of the key with 75% success rate. Whereas, our fault-assisted side-channel attacks on HMAC-Streebog, which additionally require the attacker to flip a set of chosen bits of the inner hash, can recover the key in real time. To the best of our knowledge, our passive attack is the best non-fault attack on HMAC-Streebog-256. Our fault-assisted side-channel attacks have a larger temporal window for fault injection, target a more accessible location and cannot be mitigated with output redundancy countermeasures when compared to the other fault attacks on HMAC-Streebog.

We have also devoted a chapter of this thesis to the study of securing software implementations of a family of symmetric-key ciphers. Even though the stream ciphers RCR-64 and RCR-32 have remained unbroken since they were published in 2007, we are the first to analyse their security and performance in such a comprehensive detail, to the best of our knowledge. The RCR ciphers are designed to support a wide range of key/IV sizes, without any specific recommendation from the designers. We have shown that the RCR ciphers are best used with 256-bit keys and 384-bit IVs. Based on our thorough analysis, the ciphers, especially with the recommended input sizes, are found to be secure against differential, linear, algebraic, cube, time-memory-data trade-off and distinguishing attacks. We have also suggested ways to protect software implementations of the RCR ciphers against (cache-)timing and some prominent processor flag attacks. We have also presented the performance evaluation of the ciphers, which proves that the protected implementation of the RCR-64 encrypts long messages at speeds comparable to some of the fastest stream ciphers available today. Our results highlight that the RCR ciphers may be well suited for PC-based applications in general and streaming audio / video applications in particular.

7.2 Future Research

To conclude this Chapter, we list some problems for future work that arise in connection with the topics discussed in this thesis.

Cryptanalysis. The design methodologies of cryptographic algorithms and the attack techniques to counter them are continuously evolving. The healthy conflicts between cryptographers and cryptanalysts open up a never-ending space of research problems. Since the start of this doctoral research, several new symmetric-key algorithms have been designed. Notable examples include the finalists of the NIST's lightweight cryptography standardisation process such as ASCON [74], Elephant [271], GIFT-COFB [272], Grain128-AEAD [77], ISAP [273], Photon-Beetle [274], Romulus [275], Sparkle [276], TinyJambu [277], and Xoodyak [278]. New or improved attacks on these authenticated encryption algorithms would be well-received by the community.

Processor flag attacks. The implementation of a symmetric-key algorithm in software can affect various processor flags — not limited to carry and overflow flags — depending on the underlying operations it performs. Therefore, exploring ways to exploit different flags to attack cryptographic implementations makes an interesting topic for future research. Despite being introduced about 20 years ago, this class of side-channel attacks seems to have received less attention. To evaluate the impact of these attacks, the study of side-channel techniques used to obtain processor flags calls for more attention.

Protected implementations. The study of different techniques that can be used to protect cryptographic implementations from side-channel attacks is another useful direction to proceed. It would also be worthwhile to build secure implementations of some of the popular symmetric-key algorithms using these techniques and evaluate their performances.

Design of lightweight cryptographic algorithms. Lightweight symmetric-key ciphers are gaining popularity due to their use in constrained pervasive devices, and many new designs have been proposed recently. As mentioned earlier, NIST is in the process of

identifying a lightweight authenticated encryption standard. Considering its present stature, the study and development of lightweight primitives would be an interesting direction to proceed.

Bibliography

[1] "Cryptography", Rivest, R.L., *Handbook of Theoretical Computer Science, Chapter 13*, **1990**.

[2] "Digital Signatures and Public-Key Cryptosystems", Rivest, R.L., Shamir, A., Adleman, L.M., *Commun. ACM*, **1978**, *21(2)*, 120–126.

[3] "Use of Elliptic Curves in Cryptography", Miller, S.V., *Proceedings of CRYPT0 '85, LNCS*, **1986**, *218*, 417–426.

[4] "Elliptic curve cryptosystems", Koblitz, N., *Mathematics of Computation*, **1987**, *48*, 203–209.

[5] "How to Swindle Rabin", Yuval, G., *Cryptologia*, **1979**, *3*, 187–189.

[6] "The MD5 Message Digest Algorithm", Rivest, R.L., *RFC 1321*, **1992**, available at: `http://www.ietf.org/rfc/rfc1321.txt`.

[7] "Secure Hash Standard (SHS)", Barker, E., *Federal Inf. Process. Stds. (NIST FIPS)*, **1995**, *180-1*.

[8] "Secure Hash Standard (SHS)", Dang, Q., *Federal Inf. Process. Stds. (NIST FIPS)*, **2012**, *180-4*.

[9] "The WHIRLPOOL Hashing Function", Barreto, P.S.L.M., Rijmen, V., *First open NESSIE Workshop, Leuven*, **2000**.

[10] "GOST R 34.11-2012: Hash Function", Dolmatov, V., Degtyarev, A., *RFC 6986*, **2013**, available at: `http://tools.ietf.org/html/rfc6986`.

[11] "Secrecy, authentication, and public key systems", Merkle, R.C., *PhD thesis, Stanford University*, **1979**.

[12] "A Design Principle for Hash Functions", Damågard, I., *Proceedings of CRYPT0 '89, LNCS*, **1990**, *435*, 416–427.

[13] "New Directions in Cryptography", Diffie, W., Hellman, M., *IEEE Transactions on Information Theory*, **1976**, *IT-22(6)*, 644–654.

[14] "Communication Theory of Secrecy Systems", Shannon, C.E., *Bell System Technical Journal*, **1949**, *28(4)*, 656-–715.

[15] "Analysis of Certain Aspects of Output Feedback Mode", Jueneman, R.R., *Proceedings of CRYPT0 '82, LNCS*, **1983**, 99–127.

[16] "Bluetooth Core Specification", Bluetooth™, *Bluetooth Specification, version 5.2*, **2019**, available at: `http://www.bluetooth.org`.

[17] "A pedagogical implementation of A5/1", Briceno, M., Goldberg, I., Wagner, D., *GSM A5 Files Published on Cryptome*, **2009**, available at: `http://cryptome.org/jya/a51-pi.htm`.

[18] "A New Version of the Stream Cipher SNOW", Ekdahl, P., Johansson, T., *Proceedings of SAC 2002, LNCS*, **2003**, *2595*, 47–61.

[19] "Information technology – Security techniques – Encryption algorithms – Part 4: Stream ciphers", ISO/IEC 18033-4:2011, *International Organization for Standardization*, **2011**, available at: `https://www.iso.org/standard/54532.html`.

[20] "Specification of the 3GPP Confidentiality and Integrity Algorithms UEA2 & UIA2. Document 2: SNOW 3G Specification, version 1.1", *ETSI / SAGE*, **2006**,

available at: `https://www.gsma.com/aboutus/wp-content/uploads/2014/12/snow3gspec.pdf`.

[21] "RC4", *Wikipedia*, **2021**, available at: `https://en.wikipedia.org/wiki/RC4`.

[22] "Spritz - a spongy RC4-like stream cipher and hash function", Rivest, R., Schuldt, J., **2014**, available at: `https://people.csail.mit.edu/rivest/pubs/RS14.pdf`.

[23] "A New Stream Cipher HC-256", Wu, H., *Proceedings of FSE 2004, LNCS*, **2004**, *3017*, 226–244.

[24] "The Stream Cipher HC-128", Wu, H., *New Stream Cipher Designs, LNCS*, **2008**, *4986*, 39–47.

[25] "Py (Roo): A Fast and Secure Stream Cipher using Rolling Arrays", Biham, E., Seberry, J., *Report 2005/023, eSTREAM, ECRYPT Stream Cipher Project*, **2005**.

[26] "Helix: Fast Encryption and Authentication in a Single Cryptographic Primitive", Ferguson, N., Whiting, D., Schneier, B., Kelsey, J., Lucks, S., Kohno, T., *Proceedings of FSE 2003, LNCS*, **2003**, *2887*, 330–346.

[27] "Phelix: Fast Encryption and Authentication in a Single Cryptographic Primitive", Whiting, D., Schneier, B., Lucks, S., Muller, F., *Report 2005/027, eSTREAM, ECRYPT Stream Cipher Project*, **2005**.

[28] "Primitive Specification for SSS", Rose, G., Hawkes, P., Paddon, M., de Vries, M.W., *Report 2005/028, eSTREAM, ECRYPT Stream Cipher Project*, **2005**.

[29] "The Self-synchronizing Stream Cipher Moustique", Daemen, J., Kitsos, P., *New Stream Cipher Designs, LNCS*, **2008**, *4986*, 210–223.

[30] "Announcing the Advanced Encryption Standard (AES)", Dworkin, M. , Barker, E. , Nechvatal, J. , Foti, J. , Bassham, L. , Roback, E., Dray, J., *Federal Inf. Process. Stds. (NIST FIPS)*, **2001**, *197*.

[31] "Hash Functions and Data Integrity", Menezes, A.J., Oorschot, P.C.V., Vanstone, S.A., *Handbook of Applied Cryptography*, **1997**.

[32] "Generating strong one-way functions with cryptographic algorithm", Matyas, S.M., Meyer, C.H., Oseas, J., *IBM Technical Disclosure Bulletin*, **1985**, *27*, 5658–5659.

[33] "Analysis and Design of Cryptographic Hash Functions", Preneel, B., *PhD thesis, Katholieke Universiteit Leuven*, **1993**.

[34] "128-bit hash function (N-hash)", Miyaguchi, S., Ohta, K., Iwata, M., *NTT Review*, **1990**, *2*, 128–132.

[35] "Cryptography and Computer Privacy", Feistel, H., *Scientific American*, **1973**, *228(5)*, 15–23.

[36] "DATA ENCRYPTION STANDARD (DES)", National Institute of Standards and Technology, *Federal Inf. Process. Stds. (NIST FIPS)*, **1999**, *46-3*.

[37] "How to Construct Pseudorandom Permutations from Pseudorandom Functions", Luby, M., Rackoff, C., *SIAM J. Comput.*, **1988**, *17(2)*, 373–386.

[38] "Description of a New Variable-Length Key, 64-bit Block Cipher (Blowfish)", Schneier, B., *Proceedings of FSE 1993, LNCS*, **1993**, *809*, 191–204.

[39] "A Description of the Camellia Encryption Algorithm", Matsui, M., Nakajima, J., Moriai, S., *RFC 3713*, **2004**, available at: `http://tools.ietf.org/html/rfc3713`.

[40] "3G Security; Specification of the 3GPP confidentiality and integrity algorithms; Document 2: Kasumi specification", *ETSI / SAGE*, **1999**, available at: `https://portal.etsi.org/new3g/TB/other/algorithms/35202-311.pdf`.

[41] "GOST R 34.12-2015: Block Cipher "Magma"", Dolmatov, V., Baryshkov, D., *RFC 8891*, **2015**, available at: `https://www.ietf.org/rfc/rfc8891.pdf`.

[42] "Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher", Barker, E., Mouha, N., *NIST Special Publication 800-67 Rev. 2*, **2017**.

[43] "DES Modes of Operation", National Bureau of Standards, *Federal Inf. Process. Stds. (NIST FIPS)*, **1980**, *81*.

[44] "Recommendation for Block Cipher Modes of Operation", Dworkin, M., *NIST Special Publication 800-38A*, **2001**.

[45] "OCB: A block-cipher mode of operation for efficient authenticated encryption", Rogaway, P., Bellare, M., Black, J., *ACM Trans. Inf. Syst. Secur.*, **2003**, *6(3)*, 365–403.

[46] "Counter with CBC-MAC (CCM)", Whiting, D., Housley, R., Ferguson, N., *RFC 3610*, **2003**, available at: `http://tools.ietf.org/html/rfc3610`.

[47] "The EAX mode of operation", Bellare, M., Rogaway, P., Wagner, D., *Proceedings of FSE 2004, LNCS*, **2004**, *3017*, 389–407.

[48] "The Security and Performance of the Galois/Counter Mode (GCM) of Operation", McGrew, D.A., Viega, J., *Proceedings of INDOCRYPT 2004, LNCS*, **2004**, *3348*, 343–355.

[49] "Introduction", Knudsen, L.R., Robshaw, M.J.B., *The Block Cipher Companion, Information Security and Cryptography*, **2011**, 1–12.

[50] "A Message Authenticator Algorithm Suitable for a Mainframe Computer", Davies, D., *Proceedings of CRYPTO'84, LNCS*, **1985**, *196*, 393–400.

[51] "Banking — Approved algorithm for message authentication — Part 2: Message authenticator algorithms", ISO/IEC 8731-2:1992, *International Organization for Standardization*, **1992**, available at: `https://www.iso.org/standard/16139.html`.

[52] “Information technology — Security techniques — Message Authentication Codes (MACs) — Part 1: Mechanisms using a block cipher”, ISO/IEC 9797-1:2011, *International Organization for Standardization*, **2011**, available at: `https://www.iso.org/standard/50375.html`.

[53] “CBC MAC for Real-Time Data Sources”, Petrank, E., Rackoff, C., *Journal of Cryptology*, **2000**, *13*, 315–338.

[54] “Fast Encryption and Authentication: XCBC Encryption and XECB Authentication Modes”, Gligor V.D., Donescu P., *Proceedings of FSE 2001, LNCS*, **2002**, *2355*, 92–108.

[55] “OMAC: One-Key CBC MAC”, Iwata, T., Kurosawa, K., *Proceedings of FSE 2003, LNCS*, **2003**, *2887*, 129–153.

[56] “Recommendation for Block Cipher Modes of Operation: The CMAC Mode for Authentication”, Dworkin, M., *NIST Special Publication 800-38B*, **2005**.

[57] “Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC”, Dworkin, M., *NIST Special Publication 800-38D*, **2007**.

[58] “A Block-Cipher Mode of Operation for Parallelizable Message Authentication”, Black, J., Rogaway, P., *Proceedings of EUROCRYPT 2002, LNCS*, **2002**, *2332*, 384–397.

[59] “Message authentication with one-way hash functions”, Tsudik, G., *ACM Computer Communications Review*, **1992**, *22(5)*, 29–38.

[60] “MDx-MAC and Building Fast MACs from Hash Functions”, Preneel, B., van Oorschot, P.C., *Proceedings of CRYPTO’95, LNCS*, **1995**, *963*, 1–14.

[61] “Keying Hash Functions for Message Authentication”, Bellare, M., Canetti, R., Krawczyk, H., *Proceedings of CRYPTO’96, LNCS*, **1996**, *1109*, 1–15.

[62] “The Keyed-Hash Message Authentication Code (HMAC)”, National Institute of Standards and Technology, *Federal Inf. Process. Stds. (NIST FIPS)*, **2008**, *198-1*.

[63] “Information technology — Security techniques — Message Authentication Codes (MACs) — Part 2: Mechanisms using a dedicated hash-function”, ISO/IEC 9797-2:2011, *International Organization for Standardization*, **2011**, available at: `https://www.iso.org/standard/51618.html`.

[64] “Using HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512 with IPsec”, Kelly, S., Frankel, S., *RFC 4868*, **2007**, available at: `http://tools.ietf.org/html/rfc4868`.

[65] “SHA-2 Data Integrity Verification for the Secure Shell (SSH) Transport Layer Protocol”, Bider, D., Baushke, M., *RFC 6668*, **2012**, available at: `http://tools.ietf.org/html/rfc6668`.

[66] “The Transport Layer Security (TLS) Protocol Version 1.3”, Rescorla, E., *RFC 8446*, **2018**, available at: `http://tools.ietf.org/html/rfc8446`.

[67] “Using Encryption for Authentication in Large Networks of Computers”, Needham, R.M., Schroeder, M.D., *Communications of the ACM*, **1978**, *21(12)*, 993–999.

[68] “Problem areas for the IP security protocols”, Bellovin, S., *Proceedings of USENIX Security ’96*, **1996**, 1–16.

[69] “Side-Channel Attacks on Symmetric Encryption Schemes: The Case for Authenticated Encryption”, Black, J., Urtubia, H., *Proceedings of USENIX Security ’02*, **2002**, 327–338.

[70] “Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm”, Bellare, M., Namprempre, C., *Journal of Cryptology*, **2008**, 21, 469–491.

[71] "Security Flaws Induced by CBC Padding — Applications to SSL, IPSEC, WTLS …", Vaudenay, S., *Proceedings of EUROCRYPT 2002, LNCS*, **2002**, *2332*, 534–545.

[72] "Encryption Modes with Almost Free Message Integrity", Jutla, C.S., *Journal of Cryptology*, **2008**, *21*, 5470–578.

[73] "Grain-128a: a new version of Grain-128 with optional authentication", Ågren, M., Hell, M., Johansson, T., Meier, W., *International Journal of Wireless and Mobile Computing*, **2011**, *5(1)*, 48–59.

[74] "Ascon v1.2: Lightweight Authenticated Encryption and Hashing", Dobraunig, C., Eichlseder, M., Mendel, F., Schläffer, M., *Journal of Cryptology*, **2021**, *34*, 3.

[75] "ACORN: A Lightweight Authenticated Cipher (v3)", Wu, H., *CAESAR competition*, **2016**, available at: `https://competitions.cr.yp.to/round3/acornv3.pdf`.

[76] "AEGIS: A Fast Authenticated Encryption Algorithm (v1.1)", Wu, H., Preneel, B., *CAESAR competition*, **2016**, available at: `https://competitions.cr.yp.to/round3/aegisv11.pdf`.

[77] "Grain-128AEADv2 - A lightweight AEAD stream cipher", Hell, M., Johansson, T., Maximov, A., Meier, W., Sönnerup, J., Yoshida, H., *NIST*, **2021**, available at: `https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/grain-128aead-spec-final.pdf`.

[78] "A Mathematical Theory of Communication", Shannon, C.E.,, *Bell System Technical Journal*, **1948**, *27(3)*, 379–423.

[79] "Cramming more components onto integrated circuits", Moore, G.E., *Electronics*, **1965**, *38(8)*.

[80] "Transitioning the Use of Cryptographic Algorithms and Key Lengths", Barker, E., Roginsky, A., *NIST Special Publication 800-131A Revision 2*, **2019**.

[81] "Recommendation for Random Number Generation Using Deterministic Random Bit Generators", Barker, E., Kelsey, J., *NIST Special Publication 800-90A Revision 1*, **2015**.

[82] "Recommendation for the Entropy Sources Used for Random Bit Generation", Turan, M.S., Barker, E., Kelsey, J., McKay, K., Baish, M., Boyle, M., *NIST Special Publication 800-90B*, **2018**.

[83] "Grain: a stream cipher for constrained environments", Hell, M., Johansson, T., Meier, W., *Int. J. Wire. Mob. Comput.*, **2007**, *2(1)*, 86–93.

[84] "The stream cipher MICKEY 2.0", Babbage, S., Dodd, M., *ECRYPT Stream Cipher Project Report*, **2006**, available at: `http://www.ecrypt.eu.org/stream/p3ciphers/mickey/mickey_p3.pdf`.

[85] "ChaCha, a variant of Salsa20", Bernstein, D.J., *Workshop Record of SASC 2008, The State of the Art of Stream Ciphers*, **2008**, 273–278.

[86] "The Salsa20 Family of Stream Ciphers", Bernstein, D.J., *New Stream Cipher Designs, LNCS*, **2008**, *4986*, 84–97.

[87] "KLEIN: A New Family of Lightweight Block Ciphers", Gong, Z., Nikova, S., Law, Y.W., *Proceedings of RFIDSec 2011, LNCS*, **2012**, *7055*, 1–18.

[88] "mCrypton – A Lightweight Block Cipher for Security of Low-Cost RFID Tags and Sensors", Lim, C.H., Korkishko, T., *Proceedings of WISA 2005, LNCS*, **2006**, *3786*, 243–258.

[89] "PRESENT: An Ultra-Lightweight Block Cipher", Bogdanov, A., Knudsen, L.R., Leander, G., Paar, C., Poschmann, A., Robshaw, M.J.B., Seurin, Y., Vikkelsoe, C., *Proceedings of CHES 2007, LNCS*, **2007**, *4727*, 450–466.

[90] "Information security — Lightweight cryptography — Part 2: Block ciphers", ISO/IEC 29192-2:2019, *International Organization for Standardization*, **2019**, available at: `https://www.iso.org/standard/78477.html`.

[91] "The 128-Bit Blockcipher CLEFIA", Katagi, M., Moriai, S., *RFC 6114*, **2011**, available at: `http://www.ietf.org/rfc/rfc6114.txt`.

[92] "HIGHT: A New Block Cipher Suitable for Low-Resource Device", Hong, D., Sung, J., Hong, S., Lim, J., Lee, S., Koo, BS., Lee, C., Chang, D., Lee, J., Jeong, K., Kim, H., Kim, J., Chee, S., *Proceedings of CHES 2006, LNCS*, **2006**, *4249*, 46–59.

[93] "Information technology — Security techniques — Encryption algorithms — Part 3: Block ciphers", ISO/IEC 18033-3:2010, *International Organization for Standardization*, **2010**, available at: `https://www.iso.org/standard/54531.html`.

[94] "Piccolo: An Ultra-Lightweight Blockcipher", Shibutani, K., Isobe, T., Hiwatari, H., Mitsuda, A., Akishita, T., Shirai, T., *Proceedings of CHES 2011, LNCS*, **2011**, *6917*, 342–357.

[95] "LBlock: A Lightweight Block Cipher", Wu, W., Zhang, L., *Proceedings of ACNS 2011, LNCS*, **2011**, *6715*, 327–344.

[96] "SIMON and SPECK: Block Ciphers for the Internet of Things", Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., Wingers, L., *NIST Lightweight Cryptography Workshop*, **2015**, available at: `https://csrc.nist.gov/csrc/media/events/lightweight-cryptography-workshop-2015/documents/papers/session1-shors-paper.pdf`.

[97] "Triathlon of Lightweight Block Ciphers for the Internet of Things", Dinu, D.D., Le Corre, Y., Khovratovich, D., Perrin, L., Großschädl, J., Biryukov, A., *NIST Workshop on Lightweight Cryptography 2015*, **2015**.

[98] "Chaskey: An Efficient MAC Algorithm for 32-bit Microcontrollers", Mouha, N., Mennink, B., Herrewege, A., Watanabe, D., Preneel, B., Verbauwhede, I., *Proceedings of SAC 2014, LNCS*, **2014**, *8781*, 306–323.

[99] "LEA: A 128-Bit Block Cipher for Fast Encryption on Common Processors", Hong, D., Lee, JK., Kim, DC., Kwon, D., Ryu, K.H., Lee, DG., *Proceedings of WISA 2013, LNCS*, **2014**, *8267*, 3–27.

[100] "The Skein hash function family", Niels, F., Lucks, S., Schneier, B., Whiting, D., Bellare, M., Kohno, T., Callas, J., Walker, J., *Submission to NIST (round 3)*, **2010**.

[101] "The Hash Function BLAKE", Aumasson, J.P., Meier, W., Phan, R.C.W., Henzen, L., *Information Security and Cryptography*, **2014**.

[102] "A Practical Attack on Broadcast RC4", Mantin, I., Shamir, A., *Proceedings of FSE 2001, LNCS*, **2002**, *2355*, 152–164.

[103] "Algebraic Attacks on Stream Ciphers with Linear Feedback", Courtois, N.T., Meier, W., *Proceedings of EUROCRYPT 2003, LNCS*, **2003**, *2656*, 345–359.

[104] "Fast Algebraic Attacks on Stream Ciphers with Linear Feedback", Courtois, N.T., *Proceedings of CRYPTO 2003, LNCS*, **2003**, *2729*, 176–194.

[105] "Algebraic Attacks on Combiners with Memory", Armknecht, F., Krause, M., *Proceedings of CRYPTO 2003, LNCS*, **2003**, *2729*, 162–175.

[106] "Improving Fast Algebraic Attacks", Armknecht, F., *Proceedings of FSE 2004, LNCS*, **2004**, *3017*, 65–82.

[107] "Decrypting a Class of Stream Ciphers Using Ciphertext Only", Siegenthaler, T., *Trans. on Computers*, **1985**, *C-34(1)*, 81–85.

[108] "Differential cryptanalysis of DES-like cryptosystems", Biham, E., Shamir, A., *Journal of Cryptology*, **1991**, *4*, 3–72.

[109] "Differential Cryptanalysis in Stream Ciphers", Biham, E., Dunkelman, O., *Cryptology ePrint archive*, **2007**, available at: `https://eprint.iacr.org/2007/218.pdf`.

[110] "Higher Order Derivatives and Differential Cryptanalysis", Lai, X., *Communications and Cryptography*, **1994**, *276*, 227–233.

[111] "Truncated and higher order differentials", Knudsen, L.R., *Proceedings of FSE 1994, LNCS*, **1995**, *1008*, 196–211.

[112] "Cryptanalysis of Skipjack Reduced to 31 Rounds Using Impossible Differentials", Biham, E., Biryukov, A., Shamir, A., *Proceedings of EUROCRYPT 1999, LNCS*, **1999**, *1592*, 12–23.

[113] "The Boomerang Attack", Wagner, D., *Proceedings of FSE 1999, LNCS*, **1999**, *1636*, 156–170.

[114] "A New Method for Known Plaintext Attack of FEAL Cipher", Matsui, M., Yamagishi, A., *Proceedings of EUROCRYPT 1992, LNCS*, **1993**, *658*, 81–91.

[115] "Linear Cryptanalysis Method for DES Cipher", Matsui, M., *Proceedings of EUROCRYPT 1993, LNCS*, **1994**, *765*, 386–397.

[116] "Linear cryptanalysis of stream ciphers", Golić, J.D., *Proceedings of FSE 1994, LNCS*, **1995**, *1008*, 154–169.

[117] "Integral Cryptanalysis", Knudsen, L., Wagner, D., *Proceedings of FSE 2002, LNCS*, **2002**, *2365*, 112–127.

[118] "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems", Kocher, P.C., *Proceedings of CRYPTO 1996, LNCS*, **1996**, *1109*, 104–113.

[119] "Differential Power Analysis", Kocher, P., Jaffe, J., Jun, B., *Proceedings of CRYPTO 1999, LNCS*, **1999**, *1666*, 388–397.

[120] "The EM Side-Channel(s)", Agrawal, D., Archambeault, B., Rao, J.R., Rohatgi, P., *Proceedings of CHES 2002, LNCS*, **2003**, *2523*, 29–45.

[121] "Acoustic cryptanalysis: On nosy people and noisy machines", Shamir, A., Tromer, E., *EUROCRYPT 2004 Rump Session*, **2004**, available at: `http://cs.tau.ac.il/~tromer/acoustic/ec04rump`.

[122] "Acoustic Cryptanalysis", Genkin, D., Shamir, A., Tromer, E., *Journal of Cryptology*, **2017**, *30*, 392–443.

[123] "Low temperature data remanence in static RAM", Skorobogatov, S., *University of Cambridge Technical Report 536*, **2002**, available at: `https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-536.pdf`.

[124] "Lest we Remember: Cold-boot Attacks on Encryption Keys", Halderman, J.A., Schoen, S.D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J.A., Feldman, A.J., et al., *Communications of the ACM*, **2009**, *52(5)*, 91–98.

[125] "Meltdown: Reading Kernel Memory from User Space", Lipp, M., Schwarz, M., Gruss, D., Prescher, T., Haas, W., Fogh, A., Horn, J., et al., *Proceedings of USENIX Security '18*, **2018**, 973–990.

[126] "Spectre Attacks: Exploiting Speculative Execution", Kocher, P., Horn, J., Fogh, A., Genkin, D., Gruss, D., Haas, W., Hamburg, M., et al., *IEEE Proceedings of SP 2019*, **2019**, 19–37.

[127] "Let's Not Speculate: Discovering and Analyzing Speculative Execution Attacks", Mambretti, A., Neugschwandtner, M., Sorniotti, A., Kirda, E., Robertson, W., Kurmus, A., *IBM Research Report*, **2018**, *RZ 3933*, 19–37, available at: `https://domino.research.ibm.com/library/cyberdig.nsf/papers/D66E56756964D8998525835200494B74`.

[128] “RAMBleed: Reading Bits in Memory Without Accessing Them”, Kwong, A., Genkin, D., Gruss, D., Yarom, Y., *IEEE Proceedings of SP 2020*, **2020**, 310–326.

[129] “Side Channel Cryptanalysis of Product Ciphers”, Kelsey, J., Schneier, B., Wagner, D., Hall, C., *Journal of Computer Security*, **2000**, *8(2,3)*, 141–158.

[130] “Atmel AVR 32 Architecture Document”, Atmel, **2011**, available at: `http://ww1.microchip.com/downloads/en/DeviceDoc/doc32000.pdf`.

[131] “Side Channel Cryptanalysis of Streebog”, Sekar, G., *Proceedings of SSR 2015, LNCS*, **2015**, *9497*, 154–162.

[132] “Fault Attack on Schnorr Based Identification and Signature Schemes”, Fouque, P., Masgana, D., Valette, F., *IEEE Proceedings of FDTC 2009*, **2009**, 32–38.

[133] “The Carry Leakage on the Randomized Exponent Countermeasure”, Fouque, P., Réal, D., Valette, F., Drissi, M., *Proceedings of CHES 2008, LNCS*, **2008**, *5154*, 198–213.

[134] “Hamming Weight Attacks on Cryptographic Hardware — Breaking Masking Defense”, Gomufflkiewicz M., Kutyfflowski M., *Proceedings of ESORICS 2002, LNCS*, **2002**, *2502*, 90–103.

[135] “Introduction to Side-Channel Attacks”, Standaert, FX., *Secure Integrated Circuits and Systems*, **2010**, 27–42.

[136] “SCA Countermeasures”, Ouladj, M., Guilley, S., *Side-Channel Analysis of Embedded Systems*, **2021**, 35–46.

[137] “Cache Attacks and Countermeasures: The Case of AES”, Osvik, D.A., Shamir, A., Tromer, E., *Proceedings of CT-RSA 2006, LNCS*, **2006**, *3860*, 1–20.

[138] “Side channel analysis of SPECK”, Joseph, M., Sekar, G., Balasubramanian, R., *Journal of Computer Security*, **2020**, *28(6)*, 655–676.

[139] “A New Physical Mechanism for Soft Errors in Dynamic Memories”, May, T.C., Woods, M.H., *16th Annual Reliability Physics Symposium, IEEE*, **1978**, 33–40.

[140] “On the Importance of Eliminating Errors in Cryptographic Computations”, Boneh, D., DeMillo, R.A., Lipton, R.J., *Journal of Cryptology*, **2001**, *14(2)*, 101–119.

[141] “Injection Technologies for Fault Attacks on Microprocessors”, Barenghi, A., Bertoni, G., Breveglieri, L., Pellicioli, M., Pelosi, G., *Fault Analysis in Cryptography, Information Security and Cryptography*, **2012**, 275–293.

[142] “Lightweight stream cipher cryptosystems”, Gong, G., Aagaard, M., Fan, X., *United States Patent*, US 8,953,784 B2, **2015**, available at: `https://www.google.com/patents/US8953784`.

[143] “Information technology —- Automatic identification and data capture techniques -— Part 22: Crypto suite SPECK security services for air interface communications”, ISO/IEC 29167-22:2018, *International Organization for Standardization*, **2018**, available at: `https://www.iso.org/standard/70389.html`.

[144] “Guidelines on the Cryptographic Algorithms to Accompany the Usage of Standards GOST R 34.10-2012 and GOST R 34.11-2012”, Smyshlyaev, S., Alekseev, E., Oshkin, I., Popov, V., Leontiev, S., Podobaev, V., Belyavsky, D., *RFC 7836*, **2016**, available at: `https://tools.ietf.org/html/rfc7836`.

[145] “Related-Key Attacks on the Py-family of Ciphers and an Approach to Repair the Weaknesses”, Sekar, G., Paul, S., Preneel, B., *Proceedings of INDOCRYPT 2007, LNCS*, **2007**, *4859*, 58–72.

[146] “New Results of Related-Key Attacks on All Py-family of Stream Ciphers”, Ding, L., Guan, J., Sun, W., *Journal for Universal Computer Science*, **2012**, *18(12)*, 741–756.

[147] "On the Security of the Stream Ciphers RCR-64 and RCR-32", Joseph, M., Sekar, G., Balasubramanian, R., Venkiteswaran, G., *The Computer Journal*, **2021**, *65(12)*, 3091–3099.

[148] "A Secure RFID System for Product Anti-Counterfeiting", TechConnect World, Innovation Conference & Expo 2015, available at: `https://web.archive.org/web/20150920193338/http://www.techconnectworld.com/World2015/participate/innovation/pop.html?id=205`. Last accessed 01 Dec 2021.

[149] "“Lightweight” Security Algorithm for 4G/5G Networks", Waterloo Commericalization Office, available at: `https://uwaterloo.ca/research/waterloo-commercialization-office-watco/business-opportunities-industry/lightweight-security-algorithm-4g5g-networks`. Last accessed 01 Dec 2021.

[150] "GOST R 34.11-2012: Hash Function", Dolmatov, V., Degtyarev, A., *RFC 6986*, **2013**, available at: `https://tools.ietf.org/html/rfc6986`.

[151] "Tweaking the IV Setup of the Py Family of Stream Ciphers – The Ciphers TPy, TPypy, and TPy6", Biham, E., Seberry, J., *Report 2007/038, eSTREAM, ECRYPT Stream Cipher Project*, **2007**.

[152] "Pypy: Another Version of Py", Biham, E., Seberry, J., *Report 2006/038, eSTREAM, ECRYPT Stream Cipher Project*, **2006**.

[153] "Comparison of 256-bit stream ciphers at the beginning of 2006", Bernstein, D.J., *Workshop Record of SASC 2006, Stream Ciphers Revisited*, **2006**, 70–83.

[154] "eSTREAM - Update 1", ECRYPT Network of Excellence, *Report 2005/057, eSTREAM, ECRYPT Stream Cipher Project*, **2005**.

[155] "eBACS: ECRYPT Benchmarking of Cryptographic Systems", Bernstein, D.J., Lange, T., `https://bench.cr.yp.to`. Last accessed 22 Jun 2021.

[156] “Sosemanuk, a Fast Software-Oriented Stream Cipher”, Berbain et al., *New Stream Cipher Designs, LNCS*, **2008**, *4986*, 98–118.

[157] “D.SYM.10 The eSTREAM Portfolio in 2012”, Cid, C., Robshaw, M., *ECRYPT II, European Network of Excellence in Cryptology II*, **2012**.

[158] “Intel’s New AES Instructions for Enhanced Performance and Security”, Gueron, S., *Proceedings of FSE 2009, LNCS*, **2009**, *5665*, 51–66.

[159] “CryptMT3 Stream Cipher”, Matsumoto, M., Saito, M., Nishimura, T., Hagita, M., *New Stream Cipher Designs, LNCS*, **2008**, *4986*, 7–19.

[160] “The Advanced Encryption Standard (256-bit key) in a particular counter mode”, `https://bench.cr.yp.to/impl-stream/aes256ctr.html`. Last accessed 22 Jun 2021.

[161] “The Advanced Encryption Standard (256-bit key) in another counter mode specified by eSTREAM”, `https://bench.cr.yp.to/impl-stream/aes256estream.html`. Last accessed 22 Jun 2021.

[162] “On Matsui’s Linear Cryptanalysis”, Biham, E., *Proceedings of EUROCRYPT 1994, LNCS*, **1995**, *950*, 341–355.

[163] “A Generalization of Linear Cryptanalysis and the Applicability of Matsui’s Piling-up Lemma”, Harpes, C., Kramer, G.G., Massey, J.L., *Proceedings of EUROCRYPT 1995, LNCS*, **1995**, *921*, 24–38.

[164] “Cryptanalysis of the Full Spritz Stream Cipher”, Banik, S., Isobe, T., *Proceedings of FSE 2016, LNCS*, **2016**, *9783*, 63–77.

[165] “Improved Distinguishing Attacks on HC-256”, Sekar, G., Preneel, B., *Proceedings of IWSEC 2009, LNCS*, **2009**, *5824*, 38–52.

[166] "Binary Pseudorandom Sequences of Period $2^n - 1$ with Ideal Autocorrelation", Seon, N.J., Golomb, S.W., Gong, G., Lee, H.K., Gaal, P., *IEEE Transactions on Information Theory*, **1998**, *44(2)*, 814–817.

[167] "Cryptographic Properties of the Welch-Gong Transformation Sequence Generators", Gong, G., Youssef, A.M., *IEEE Transactions on Information Theory*, **2002**, *48(11)*, 2837–2846.

[168] "Hardware Implementations of the WG-5 Cipher for Passive RFID Tags", Aagaard, M., Gong, G., Mota, R.K., *IEEE Proceedings of HOST 2013*, **2013**, 29–34.

[169] "A Lightweight Stream Cipher WG-7 for RFID Encryption and Authentication", Luo, Y., Chai, Q., Gong, G., Lai, X., *IEEE Proceedings of GLOBECOM 2010*, **2010**, 1–6.

[170] "WG-8: A Lightweight Stream Cipher for Resource-Constrained Smart Devices", Fan, X., Mandal, K., Gong, G., *Proceedings of QSHINE 2013, LNICST*, **2013**, *115*, 617–632.

[171] "Specification of the Stream Cipher WG-16 Based Confidentiality and Integrity Algorithms", Fan, X., Gong, G., *University of Waterloo Technical Report, CACR 2013-06*, **2013**.

[172] "WG: A Family of Stream Ciphers with Designed Randomness Properties", Gong, G., Youssef, A.M., *Information Sciences*, **2008**, *178(7)*, 1903–1916.

[173] "ECRYPT: The eSTREAM Project", `https://www.ecrypt.eu.org/stream/project.html`. Last accessed 01 Dec 2021.

[174] "Resynchronization Attacks on WG and LEX", Wu, H., Preneel, B., *Proceedings of FSE 2006, LNCS*, **2006**, *4047*, 422–432.

[175] "Attacking the Filter Generator over GF(2^m)", Rønjom, S., Helleseth, T., *Proceedings of WAIFI 2007, LNCS*, **2007**, *4547*, 264–275.

[176] "Cryptanalysis of WG-7: a Lightweight Stream Cipher", Orumiehchiha, M.A., Pieprzyk, J., Steinfeld, R., *Cryptography and Communications*, **2012**, *4(3)*, 277–285.

[177] "Cryptanalysis of Lightweight WG-8 Stream Cipher", Ding, L., Jin, C., Guan, J., Wang, Q., *IEEE Transactions on Information Forensics and Security*, **2014**, *9(4)*, 645–652.

[178] "Cryptanalysis of WG Family of Stream Ciphers", Ding, L., Jin, C., Guan, J., Zhang, S., Cui, T., Han, D., Zhao, W., *The Computer Journal*, **2015**, *58(10)*, 2677–2685.

[179] "Powers of Subfield Polynomials, Cyclic Codes and Algebraic Attacks with Applications to the WG Stream Ciphers", Rønjom, S., *Proceedings of WCC 2015*, **2015**, available at: `https://hal.inria.fr/hal-01276274`.

[180] "Optimal Parameters for the WG Stream Cipher Family", Mandal, K., Gong, G., Fan, X., Aagaard, M., *Cryptography and Communications*, **2013**, *6(2)*, 117–135.

[181] "Resilience to Distinguishing Attacks on WG-7 Cipher and Their Generalizations", Gong, G., Aagaard, M., Fan, X., *Cryptography and Communications*, **2013**, *5(4)*, 277–289.

[182] "crypto: speck - add support for the Speck block cipher", `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=da7a0ab5b4babbe5d7a46f852582be06a00a28f0`. Last accessed 06 Apr 2022.

[183] "crypto: speck - remove Speck", `https://git.kernel.org/pub/scm/linux/kernel/git/herbert/cryptodev-2.6.git/commit/?anzwix=1&id=578bdaabd015b9b164842c3e8ace9802f38e7ecc`. Last accessed 06 Apr 2022.

[184] “Google Decides Not To Use Speck For Disk Encryption, Instead Developing HPolyC”, `https://www.phoronix.com/scan.php?page=news_item&px=No-Speck-Yes-HPolyC-Encryption`. Last accessed 06 Apr 2022.

[185] “Differential Cryptanalysis of Round-Reduced SIMON and SPECK”, Abed, F., List, E., Lucks, S., Wenzel, J., *Proceedings of FSE 2014, LNCS*, **2015**, *8540*, 525–545.

[186] “Differential Analysis of Block Ciphers SIMON and SPECK”, Biryukov, A., Roy, A., Velichkov, V., *Proceedings of FSE 2014, LNCS*, **2015**, *8540*, 546–570.

[187] “Improved Differential Cryptanalysis of Round-Reduced Speck”, Dinur, I., *Proceedings of SAC 2014, LNCS*, **2014**, *8781*, 147–164.

[188] “Improved Differential Fault Attack on the Block Cipher SPECK”, Tupsamudre, H., Bisht, S., Mukhopadhyay, D., *IEEE Proceedings of FDTC 2014*, **2014**, 40–48.

[189] “Differential Fault Analysis on the Families of SIMON and SPECK Ciphers”, Huo, Y., Zhang, F., Feng, X., Wang, L., *IEEE Proceedings of FDTC 2015*, **2015**, 28–34.

[190] “Automatic Search for Linear Trails of the SPECK Family”, Yuan, Y., Zhang, B., Wu, W., *IEEE Proceedings of ISC 2015, LNCS*, **2015**, *9290*, 158–176.

[191] “MILP-Based Automatic Search Algorithms for Differential and Linear Trails for Speck”, Fu, K., Wang, M., Guo, Y., Sun, S., Hu, L., *Proceedings of FSE 2016, LNCS*, **2016**, *9783*, 268–288.

[192] “Automatic Differential Analysis of ARX Block Ciphers with Application to SPECK and LEA”, Song, L., Huang, Z., Yang, Q., *Proceedings of ACISP 2016, LNCS*, **2016**, *9723*, 379–394.

[193] "Improved Fault Analysis on the Block Cipher SPECK by Injecting Faults in the Same Round", Feng, J., Chen, H., Gao, S., Fan, L., Feng, D., *Proceedings of ICISC 2016, LNCS*, **2017**, *10157*, 317–332.

[194] "Rotational-XOR Cryptanalysis of Reduced-round SPECK", Liu, Y., De Witte, G., Ranea, A., Ashur, T., *IACR Transactions on Symmetric Cryptology*, **2017**, *2017(3)*, 24–36.

[195] "Power Analysis and Protection on SPECK and Its Application in IoT", Ge, J., Wang, A., Zhu, L., Liu, X., Shang, N., Zhang, G., *Proceedings of SecureComm 2019, LNICST*, **2019**, *305*, 350–362.

[196] "Fully Automated Differential Fault Analysis on Software Implementations of Block Ciphers", Hou, X., Breier, J., Zhang, F., Liu, Y., *IACR Transactions on Cryptographic Hardware and Embedded Systems*, **2019**, *2019(3)*, 1–29.

[197] "Differential Cryptanalysis of Round-Reduced SPECK Suitable for Internet of Things Devices", Dwivedi, A.D., Morawiecki, P., Srivastava, G., *IEEE Access*, **2019**, *7*, 16476–16486.

[198] "Impossible Differential Cryptanalysis of SPECK", Li, M., Guo, J., Cui, J., Xu, L., *Proceedings of CTCIS 2018, CCIS*, **2019**, *960*, 16–31.

[199] "Cryptanalysis of Reduced-Round SPECK", Ren, J., Chen, S., *IEEE Access*, **2019**, *7*, 63045–63056.

[200] "Efficient Computation of Boomerang Connection Probability for ARX-Based Block Ciphers with Application to SPECK and LEA", Kim, D., Kwon, D., Song, J., *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, **2020**, *E103.A(4)*, 677–685.

[201] "Automatic Search for the Linear (Hull) Characteristics of ARX Ciphers: Applied to SPECK, SPARX, Chaskey, and CHAM-64", Huang, M., Wang, L., *Security and Communication Networks*, **2020**, *2020*, Article ID 4898612.

[202] "Deep Learning-Based Cryptanalysis of Lightweight Block Ciphers", Khokhar, U.M., *Security and Communication Networks*, **2020**, *2020*, Article ID 3701067.

[203] "Continuous Diffusion Analysis", Coutinho, M., De Sousa, R.T., Borges, F., *IEEE Access*, **2020**, *8*, 123735–123745.

[204] "On the Practical (In-)Security of 64-bit Block Ciphers: Collision Attacks on HTTP over TLS and OpenVPN", Bhargavan, K., Leurent, G., *Proceedings of ACM SIGSAC CCS 2016, CCIS*, **2016**, 456–467.

[205] "Notes on the design and analysis of SIMON and SPECK", Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., Wingers, L., *Cryptology ePrint archive*, **2017**, available at: `http://eprint.iacr.org/2017/560`.

[206] "Implementation and Performance of the SIMON and SPECK Lightweight Block Ciphers on ASICs", Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., Wingers, L., **2016**, available at: `https://nsacyber.github.io/simon-speck/papers/simon-speck-asic-2014.pdf`.

[207] "The SIMON and SPECK Families of Lightweight Block Ciphers", Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., Wingers, L., *Cryptology ePrint archive*, **2013**, available at: `http://eprint.iacr.org/2013/404`.

[208] "Buffer overflow attack with multiple fault injection and a proven countermeasure", Nashimoto, S., Homma, N., Hayashi, Y., Takahashi, J., Fuji, H., Aoki, T., *Journal of Cryptographic Engineering*, **2017**, *7*, 35–46.

[209] "Fault Attacks on Secure Embedded Software: Threats, Design, and Evaluation", Yuce, B., Schaumont, P., Witteman, M., *Journal of Hardware and Systems Security*, **2018**, *2*, 111–130.

[210] "The SIMON and SPECK Block Ciphers on AVR 8-Bit Microcontrollers", Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., Wingers, L., *Proceedings of LightSec 2014, LNCS*, **2015**, *8898*, 3–20.

[211] "FELICS – Fair Evaluation of Lightweight Cryptographic Systems", Dinu, D., Biryukov, A., Großschädl, J., Khovratovich, D., Corre, Y.L., Perrin, L., *Proceedings of NIST Lightweight Cryptography Workshop 2015*, **2015**, available at: `https://csrc.nist.gov/csrc/media/events/lightweight-cryptography-workshop-2015/documents/papers/session7-dinu-paper.pdf`.

[212] "SIMON and SPECK: Block Ciphers for the Internet of Things", Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., Wingers, L., *Proceedings of NIST Lightweight Cryptography Workshop 2015*, **2015**, available at: `https://csrc.nist.gov/csrc/media/events/lightweight-cryptography-workshop-2015/documents/papers/session1-shors-paper.pdf`.

[213] "Atmel AVR Instruction Set Manual [OTHER]", Atmel, **2016**, available at: `http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-0856-AVR-Instruction-Set-Manual.pdf`.

[214] "Return-Oriented Programming", Prandini, M., Ramilli, M., *IEEE Security & Privacy*, **2012**, *10(6)*, 84–87.

[215] "Return-Oriented Programming on a Cortex-M Processor", Weidler, N.R., Brown, D., Mitchel, S.A., Anderson, J., Williams, J.R., Costley, A., Kunz, C., Wilkinson, C., Wehbe, R., Gerdes, R., *Proceedings of IEEE Trustcom 2017*, **2017**, 823–832.

[216] "Return-oriented programming on a resource constrained device", Weidler, N.R., Brown, D., Mitchell, S.A., Anderson, J., Williams, J.R., Costley, A., Kunz, C., Wilkinson, C., Wehbe, R., Gerdes, R.M., *Sustainable Computing: Informatics and Systems*, **2019**, *22*, 244–256.

[217] "Localized Electromagnetic Analysis of Cryptographic Implementations", Heyszl, J., Mangard, S., Heinz, B., Stumpf, F., Sigl, G., *Proceedings of CT-RSA 2012, LNCS*, **2012**, *7178*, 231–244.

[218] "Power and electromagnetic analysis: Improved model, consequences and comparisons", Peeters, E., Standaert, F.X., Quisquater, J.J., *Integration*, **2007**, *40(1)*, 52–60.

[219] "A survey of electromagnetic side-channel attacks and discussion on their case-progressing potential for digital forensics", Sayakkara, A., Le-Khac, N.A., Scanlon, M., *Digital Investigation*, **2019**, *29*, 43–54.

[220] "Practical Electromagnetic Template Attack on HMAC", Fouque, PA., Leurent, G., Réal, D., Valette, F., *Proceedings of CHES 2009, LNCS*, **2009**, *5747*, 66–80.

[221] "HMAC: Keyed-Hashing for Message Authentication", Krawczyk, H., Bellare, M., Canetti, R., *RFC 2104*, **1997**, available at: `https://tools.ietf.org/html/rfc2104`.

[222] "Impossible Differential Properties of Reduced Round Streebog", Abdelkhalek, A., AlTawy, R., Youssef, A.M., *Proceedings of C2SI 2015, LNCS*, **2015**, *9084*, 274–286.

[223] "Watch your constants: malicious Streebog", AlTawy, R., Youssef, A.M., *IET Information Security*, **2015**, *9(6)*, 328–333.

[224] "Reverse-Engineering the S-Box of Streebog, Kuznyechik and STRIBOBr1", Biryukov, A., Perrin, L., Udovenko, A., *Proceedings of EUROCRYPT 2016, LNCS*, **2016**, *9665*, 372–402.

[225] "Exponential S-Boxes: a Link Between the S-Boxes of BelT and Kuznyechik/Streebog", Perrin, L., Udovenko, A., *IACR Transactions on Symmetric Cryptology*, **2017**, *2016(2)*, 99–124.

[226] "Partitions in the S-Box of Streebog and Kuznyechik", Perrin, L., *IACR Transactions on Symmetric Cryptology*, **2019**, *2019(1)*, 302–329.

[227] "Differential Fault Analysis of Streebog", AlTawy, R., Youssef, A.M., *Proceedings of ISPEC 2015, LNCS*, **2015**, *9065*, 35–49.

[228] "Improved Generic Attacks against Hash-Based MACs and HAIFA", Dinur, I., Leurent, G., *Proceedings of CRYPTO 2014, LNCS*, **2014**, *8616*, 149–168.

[229] "GOST R 34.11-2012 hash function with 512/256 bit digest", Degtyarev, A., **2019**, available at: `https://github.com/adegtyarev/streebog`.

[230] "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors", Kim, Y., Daly, R., Kim, J., Fallin, C., Lee, J.H., Lee, D., Wilkerson, C., Lai, K., Mutlu, O., *Proceedings of ACM/IEEE ISCA 2014*, **2014**, 361–372.

[231] "Lest We Remember: Cold Boot Attacks on Encryption Keys", Halderman, J.A., Schoen, S.D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J.A., Feldman, A.J., Appelbaum, J., Felten, E.W., *Communications of the ACM*, **2009**, *52(5)*, 91–98.

[232] "TRESOR runs encryption securely outside RAM", Müller, T., Freiling, F.C., Dewald, A., *Proceedings of USENIX Security '11*, **2011**, 251–266.

[233] "PRIME: Private RSA infrastructure for memory-less encryption", Garmany, B., Müller, T., *ACM Proceedings of ACSAC 2013*, **2013**, 149–158.

[234] "Cryptographic key protection against FROST for mobile devices", Zhang, X., Tan, Y., Xue, Y., Zhang, Q., Li, Y., Zhang, C., Zheng, J., *Cluster Computing*, **2017**, *20(3)*, 2393–2402.

[235] "ARMORED: CPU-bound encryption for android-driven ARM devices", Götzfried, J., Müller, T., *IEEE Proceedings of ARES 2013*, **2013**, 161–168.

[236] "PixelVault: Using GPUs for securing cryptographic operations", Vasiliadis, G., Athanasopoulos, E., Polychronakis, M., Ioannidis, S., *ACM Proceedings of CCS 2014*, **2014**, 1131–1142.

[237] "AESSE: A cold-boot resistant implementation of AES", Muller, T., Dewald, A., Freiling, F.C., *ACM Proceedings of EUROSEC '10*, **2010**, 42–47.

[238] "Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks", Cojocar, L., Razavi, K., Giuffrida, C., Bos, H., *IEEE Proceedings of SP 2019*, **2019**, 279–295.

[239] "ERROL - InfraRed (IR) Laser Bench Solution For Security Evaluation In 2017", Hubert, C., Côme, M., `https://www.errol-laser.com/_files/ugd/1c6d8e_1e3b60623c454f02b125c891cd53a245.pdf`. Last accessed 08 Jun 2022.

[240] "Distinguishing Attacks on the Stream Cipher Py", Paul, S., Preneel, B., Sekar, G., *Proceedings of FSE 2006, LNCS*, **2006**, *4047*, 405–421.

[241] "Improved cryptanalysis of Py", Crowley, P., *Report 2006/010, eSTREAM, ECRYPT Stream Cipher Project*, **2006**.

[242] "Differential Cryptanalysis of the Stream Ciphers Py, Py6 and Pypy", Wu, H., Preneel, B., *Proceedings of EUROCRYPT 2007, LNCS*, **2007**, *4515*, 276–290.

[243] "How to Break Py and Pypy by a Chosen-IV Attack", Isobe, T., Ohigashi, T., Kuwakado, H., Morii, M., *Report 2006/060, eSTREAM, ECRYPT Stream Cipher Project*, **2006**.

[244] "New Attacks on the Stream Cipher TPy6 and Design of New Ciphers the TPy6-A and the TPy6-B", Sekar, G., Paul, S., Preneel, B., *Proceedings of WEWoRC 2007, LNCS*, **2008**, *4945*, 127–141.

[245] "New Weaknesses in the Keystream Generation Algorithms of the Stream Ciphers TPy and Py", Sekar, G., Paul, S., Preneel, B., *Proceedings of ISC 2007, LNCS*, **2007**, *4779*, 249–262.

[246] "Weaknesses in the Pseudorandom Bit Generation Algorithms of the Stream Ciphers TPypy and TPy", Sekar, G., Paul, S., Preneel, B., *Cryptology ePrint archive*, **2007**, available at: `https://eprint.iacr.org/2007/075.pdf`.

[247] "A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications", Bassham, L., Rukhin, A., Soto, J., Nechvatal, J., Smid, M., Leigh, S., Levenson, M., Vangel, M., Heckert, N., Banks, D., *Special Publication (NIST SP 800-22 Revision 1a)*, **2010**.

[248] "Statistical Analysis of Synchronous Stream Ciphers", Turan, M.S., Doğanaksoy, A. , Çalik, C., *Workshop Record of SASC 2006, Stream Ciphers Revisited*, **2006**.

[249] "Recommendation for Key Management: Part 1 – General", Barker, E., *Special Publication (NIST SP 800-57 Part 1 Revision 5)*, **2020**.

[250] "From Schrödinger's equation to the quantum search algorithm", Grover, L.K., *Pramana - Journal of Physics*, **2001**, *56*, 333–348.

[251] "Treatment of the initial value in Time-Memory-Data Tradeoff attacks on stream ciphers", Dunkelman, O., Keller, N., *Information Processing Letters*, **2008**, *107(5)*, 133–137.

[252] "On the distribution of characteristics in bijective mappings", O'Connor, L., *Journal of Cryptology*, **1995**, *8*, 67–86.

[253] "Efficient Algorithms for Computing Differential Properties of Addition", Lipmaa, H., Moriai, S., *Proceedings of FSE 2001, LNCS*, **2002**, *2355*, 336–350.

[254] "Algebraic Attacks on SOBER-t32 and SOBER-t16 without Stuttering", Cho, J.Y., Pieprzyk, J., *Proceedings of FSE 2004, LNCS*, **2004**, *3017*, 49–64.

[255] "Advanced Algebraic Attack on Trivium", Quedenfeld, FM., Wolf, C., *Proceedings of MACIS 2015, LNCS*, **2016**, *9582*, 268–282.

[256] "Guess-and-Determine Algebraic Attack on the Self-Shrinking Generator", Cho, J.Y., Pieprzyk, J., *Proceedings of FSE 2008, LNCS*, **2008**, *5086*, 235–252.

[257] "A New Weakness in the RC4 Keystream Generator and an Approach to Improve the Security of the Cipher", Paul, S., Preneel, B., *Proceedings of FSE 2004, LNCS*, **2004**, *3017*, 245–259.

[258] "Analysis of RC4 and Proposal of Additional Layers for Better Security Margin", Maitra, S., Paul, G., *Proceedings of INDOCRYPT 2008, LNCS*, **2008**, *5365*, 27–39.

[259] "VMPC One-Way Function and Stream Cipher", Zoltak, B., *Proceedings of FSE 2004, LNCS*, **2004**, *3017*, 210–225.

[260] "ISAAC", Jenkins, R.J., *Proceedings of FSE 1996, LNCS*, **1996**, *1039*, 41–49.

[261] "A 32-bit RC4-like Keystream Generator", Nawaz, Y., Gupta, K.C., Gong, G., *Cryptology ePrint archive*, **2005**, available at: `https://eprint.iacr.org/2005/175.pdf`.

[262] "Towards a General RC4-Like Keystream Generator", Gong, G., Gupta, K.C., Hell, M., Nawaz, Y., *Proceedings of CISC 2005, LNCS*, **2005**, *3822*, 162–174.

[263] "Cube attacks on tweakable black box polynomials", Dinur, I., Shamir, A., *Proceedings of EUROCRYPT 2009, LNCS*, **2009**, *5479*, 278–299.

[264] "Improved "exhaustive search" attacks on stream ciphers", Babbage, S.H., *Proceedings of European Convention on Security and Detection*, **1995**, 161–166.

[265] "Cryptanalysis of Alleged A5 Stream Cipher", Golić, J.D., *Proceedings of EUROCRYPT 1997, LNCS*, **1997**, *1233*, 239–255.

[266] "Cryptanalytic Time/Memory/Data Tradeoffs for Stream Ciphers", Biryukov, A., Shamir, A., *Proceedings of ASIACRYPT 2000, LNCS*, **2000**, *1976*, 1–13.

[267] "New Applications of Time Memory Data Tradeoffs", Hong, J., Sarkar, P., *Proceedings of ASIACRYPT 2005, LNCS*, **2005**, *3788*, 353–372.

[268] "Distinguishing Attack Against TPypy", Tsunoo, Y., Saito, T., Kawabata, T., Nakashima, H., *Proceedings of SAC 2007, LNCS*, **2007**, *4876*, 396–407.

[269] "mebinjp/RCR_SUPERCOP: Procedure to evaluate the performance of RCR ciphers using the SUPERCOP toolkit (rcr)", Joseph, M., **2022**, available at: `https://doi.org/10.5281/zenodo.7006524`.

[270] "Information technology — Automatic identification and data capture techniques — Part 21: Crypto suite SIMON security services for air interface communications", ISO/IEC 29167-21:2018, *International Organization for Standardization*, **2018**, available at: `https://www.iso.org/standard/70388.html`.

[271] "Elephant v2", Beyne, T., Chen, Y.L., Lamarr, C.D., Mennink, B., *NIST Lightweight Cryptography Project*, **2021**, available at: `https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/elephant-spec-final.pdf`.

[272] "GIFT-COFB v1.1", Banik, S., Chakraborti, A., Iwata, T., Minematsu, K., Nandi, M., Peyrin, T., Sasaki, Y., Sim, S.M., Todo, Y., *NIST Lightweight Cryptography*

Project, **2021**, available at: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/gift-cofb-spec-final.pdf.

[273] "ISAP – Towards Side-Channel Secure Authenticated Encryption", Dobraunig, C., Eichlseder, M., Mangard, S., Mendel, F., Unterluggauer, T., *IACR Transactions on Symmetric Cryptology*, **2017**, *2017(1)*, 80–105.

[274] "PHOTON-Beetle Authenticated Encryption and Hash Family", Bao, Z., Chakraborti, A., Datta, N., Guo, J., Nandi, M., Peyrin, T., Yasuda, K., *NIST Lightweight Cryptography Project*, **2021**, available at: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/photon-beetle-spec-final.pdf.

[275] "Duel of the Titans: The Romulus and Remus Families of Lightweight AEAD Algorithms", Iwata, T., Khairallah, M., Minematsu, K., Peyrin, T., *IACR Transactions on Symmetric Cryptology*, **2020**, *2020(1)*, 43–120.

[276] "Lightweight AEAD and Hashing using the Sparkle Permutation Family", Beierle, C., Biryukov, A., Cardoso dos Santos, L., Großschädl, J., Perrin, L., Udovenko, A., Velichkov, V., Wang, Q., *IACR Transactions on Symmetric Cryptology*, **2020**, *2020(S1)*, 208–261.

[277] "TinyJAMBU: A Family of Lightweight Authenticated Encryption Algorithms (Version 2)", Wu, H., Huang, T., *NIST Lightweight Cryptography Project*, **2021**, available at: https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/tinyjambu-spec-final.pdf.

[278] "Xoodyak, a lightweight cryptographic scheme", Daemen, J., Hoffert, S., Peeters, M., Van Assche, G., Van Keer, R., *IACR Transactions on Symmetric Cryptology*, **2020**, *2020(S1)*, 60–87.