Design, analysis and implementation of hardware-oriented authenticated encryption

Khairallah, Mustafa [Mustafa Mahmoud Mohammed Kairallah]

2021

Khairallah, M. (2021). Design, analysis and implementation of hardware-oriented authenticated encryption. Singapore: Nanyang Technological University.

<https://hdl.handle.net/10356/145739>

<https://doi.org/10.32657/10356/145739>

This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License (CC BY-NC 4.0).

Downloaded on 26 Apr 2023 20:42:46 SGT

**NANYANG
TECHNOLOGICAL
UNIVERSITY**

SINGAPORE

**Design, Analysis and Implementation of
Hardware-Oriented Authenticated Encryption**

MUSTAFA KHAIRALLAH
[MUSTAFA MAHMOUD MOHAMMED KAIRALLAH]

School of Physical and Mathematical Sciences

2021

Design, Analysis and Implementation of Hardware-Oriented Authenticated Encryption

MUSTAFA KHAIRALLAH
[MUSTAFA MAHMOUD MOHAMMED KAIRALLAH]

School of Physical and Mathematical Sciences

**A thesis submitted to the Nanyang Technological University
in partial fulfilment of the requirement for the degree of
Doctor of Philosophy**

2021

Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research done by me except where otherwise stated in this thesis. The thesis work has not been submitted for a degree or professional qualification to any other university or institution. I declare that this thesis is written by myself and is free of plagiarism and of sufficient grammatical clarity to be examined. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

06/01/2021

Mustafa Mahmoud Mohammed Kairallah

Supervisor Declaration Statement

I have reviewed the content and presentation style of this thesis and declare it of sufficient grammatical clarity to be examined. To the best of my knowledge, the thesis is free of plagiarism and the research and writing are those of the candidate's except as acknowledged in the Author Attribution Statement. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

06/01/2021

Assoc. Prof. Thomas Peyrin

Authorship Attribution Statement

This thesis contains material from 5 papers published in the following peer-reviewed journals / from papers accepted at conferences in which I am listed as an author. It also contains material from 5 non-peer-reviewed technical reports. During my PhD studies I have also contributed to 12 papers/technical reports that are not part of this thesis.

Journal/Conference Papers

1. Tetsu Iwata, Mustafa Khairallah, Kazuhiko Minematsu and Thomas Peyrin. Duel of the Titans: The Romulus and Remus Families of Lightweight AEAD Algorithms. *IACR Transactions on Symmetric Cryptology*, (1), 2020.
<https://eprint.iacr.org/2019/992>.
2. Tetsu Iwata, Mustafa Khairallah, Kazuhiko Minematsu and Thomas Peyrin. Updates on Romulus, Remus and TGIF. *NIST Lightweight Cryptography Workshop*, 2019.
<https://csrc.nist.gov/CSRC/media/Events/lightweight-cryptography-workshop-2019/documents/papers/updates-on-romulus-remus-tgif-lwc2019.pdf>.
3. Mustafa Khairallah. Weak Keys in the Rekeying Paradigm: Application to COMET and mixFeed. *IACR Transactions on Symmetric Cryptology*, (4), 2019.
<https://eprint.iacr.org/2019/888>.
4. Mustafa Khairallah, Zakaria Najm, Anupam Chattopadhyay and Thomas Peyrin. Crack Me If You Can: Hardware Acceleration Bridging the Gap between Practical and Theoretical Cryptanalysis? A Survey. In *Proceedings of the 18th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, SAMOS '18*, page 167–172, New York, NY, USA, 2018. Association for Computing Machinery.
<https://dl.acm.org/doi/abs/10.1145/3229631.3239366>.
5. Mustafa Khairallah, Anupam Chattopadhyay, and Thomas Peyrin. Looting the LUTs: FPGA Optimization of AES and AES-like Ciphers for Authenticated Encryption. In A. Patra and N. P. Smart, editors, *Progress in Cryptology - INDOCRYPT 2017*, pages 282–301, Cham, 2017. Springer International Publishing.
<https://eprint.iacr.org/2017/1019.pdf>.

I had the leading role in [4, 5]. I am the sole author of [3]. [1, 2] are collaborative projects where I worked on algorithm design, circuit optimization and both hardware and software implementations. Tetsu Iwata and Kazuhiko Minematsu worked on algorithm design and security proofs. Thomas Peyrin worked on cipher design, algorithm design and led the project.

The introduction of Chapter 3 is based on [4]. Chapter 4 is based on [5]. Most of Chapter 6 is based on [3]. Chapter 7 is based on [1, 2].

Technical Reports

1. Tetsu Iwata, Mustafa Khairallah, Kazuhiko Minematsu, Thomas Peyrin, Yu Sasaki, Siang Meng Sim and Sun Ling. Thank Goodness It's Friday (TGIF). *NIST Lightweight Cryptography Project, 2019.*
<https://csrc.nist.gov/Projects/Lightweight-Cryptography/Round-1-Candidates>.
2. Tetsu Iwata, Mustafa Khairallah, Kazuhiko Minematsu and Thomas Peyrin. Romulus. *NIST Lightweight Cryptography Project, 2019.*
<https://csrc.nist.gov/Projects/Lightweight-Cryptography/Round-2-Candidates>.
3. Tetsu Iwata, Mustafa Khairallah, Kazuhiko Minematsu and Thomas Peyrin. REMUS. *NIST Lightweight Cryptography Project, 2019.*
<https://csrc.nist.gov/Projects/Lightweight-Cryptography/Round-1-Candidates>.
4. Mustafa Khairallah. Forgery Attack on mixFeed in the Nonce-Misuse Scenario. *Cryptology ePrint Archive, Report 2019/457, 2019.*
<https://eprint.iacr.org/2019/457>.
5. Sachin Kumar, Jawad Haj-Yahya, Mustafa Khairallah, Mahmoud A. Elmoher and Anupam Chattopadhyay. A Comprehensive Performance Analysis of Hardware Implementations of CAESAR Candidates. *Cryptology ePrint Archive, Report 2017/1261, 2017.*
<https://eprint.iacr.org/2017/1261>.

I am the sole author of [4]. [2, 3] are collaborative projects where I worked on algorithm design, circuit optimization and both hardware and software implementations. Tetsu Iwata and Kazuhiko Minematsu worked on algorithm design and security proofs. Thomas Peyrin worked on cipher design, algorithm design and led the project. [5] was led by Sachin Kumar. [1] is a collaboration between the team of [3] and the team composed of Thomas Peyrin, Ling Sun and Yu Sasaki who designed the TGIF cipher.

[5] is used to provide arguments for Chapters 4 and 7. Section 6.5 is based on [4]. Chapter 7 is based on [1, 2, 3].

Additional Papers

1. Jakub Breier, Mustafa Khairallah, Xiaolu Hou and Yang Liu. A Countermeasure Against Statistical Ineffective Fault Analysis. *IEEE Transactions on Circuits and Systems II: Express Briefs, 2020.*
<https://eprint.iacr.org/2019/515>.
2. Vinay B. Y. Kumar, Suman Deb, Rupesh Kumar, Mustafa Khairallah, Anupam Chattopadhyay and Avi Mendelson. Recruiting Fault Tolerance Techniques for Microprocessor Security. In *2019 IEEE 28th Asian Test Symposium (ATS)*, pages 80–805, Dec 2019.
<https://ieeexplore.ieee.org/abstract/document/8949402>.

3. Sumanta Sarkar, Mustafa Khairallah and Raghvendra Rohit. Round 2 Official Comment: ORANGE. *NIST Lightweight Cryptography Forum*, 2019.
<https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/officialcomments/orange-round2-official-comment.pdf>
4. Prasanna Ravi, Zakaria Najm, Shivam Bhasin, Mustafa Khairallah, Sourav Sen Gupta and Anupam Chattopadhyay. Security is an architectural design constraint. *Microprocessors and Microsystems*, 68:17–27, 2019.
<https://www.sciencedirect.com/science/article/abs/pii/S0141933118302229>
5. Mustafa Khairallah, Xiaolu Hou, Zakaria Najm, Jakub Breier, Shivam Bhasin and Thomas Peyrin. SoK: On DFA Vulnerabilities of Substitution-Permutation Networks. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security, Asia CCS '19*, page 403–414, New York, NY, USA, 2019. Association for Computing Machinery.
<https://eprint.iacr.org/2019/387.pdf>
6. Mustafa Khairallah, Shivam Bhasin, Anupam Chattopadhyay. On Misuse of Nonce-Misuse Resistance : Adapting Differential Fault Attacks on (few) CAESAR Winners. In *2019 IEEE 8th International Workshop on Advances in Sensors and Interfaces (IWASI)*, pages 189–193, June 2019.
<https://eprint.iacr.org/2019/585.pdf>
7. Mustafa Khairallah, Shivam Bhasin and Karim M. Abdellatif. On Comparison of Countermeasures against Statistical Ineffective Fault Attacks. In *2019 31st International Conference on Microelectronics (ICM)*, pages 122–125, Dec 2019.
<https://ieeexplore.ieee.org/document/9021933>
8. Mustafa Khairallah. Forgery Attack on SNEIKEN. *Cryptology ePrint Archive, Report 2019/408*, 2019.
<https://eprint.iacr.org/2019/408>
9. Mustafa Khairallah, Rajat Sadhukhan, Radhamanjari Samanta, Jakub Breier, Shivam Bhasin, Rajat S. Chakraborty, Anupam Chattopadhyay and Debdeep Mukhopadhyay. DFARPA: Differential fault attack resistant physical design automation. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1171–1174, March 2018.
<https://ieeexplore.ieee.org/document/8342190>
10. Mustafa Khairallah, Anupam Chattopadhyay, Bimal Mandal and Subhamoy Maitra. On Hardware Implementation of Tang-Maitra Boolean Functions. In *L. Budaghyan and F. Rodríguez-Henríquez, editors, Arithmetic of Finite Fields*, pages 111–127, Cham, 2018. Springer International Publishing.
<https://eprint.iacr.org/2018/667.pdf>
11. Mahmoud A. Elmoehr, Sachin Kumar, Mustafa Khairallah and Anupam Chattopadhyay. A Hardware-Efficient Implementation of CLOC for On-chip Authenticated Encryption. In *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 311–315, July 2018.
<https://ieeexplore.ieee.org/document/8429385>

12. Anubhab Baksi, Shivam Bhasin, Jakub Breier, Mustafa Khairallah and Thomas Peyrin. Protecting block ciphers against differential fault attacks without re-keying. *In 2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 191–194, April 2018.
<https://eprint.iacr.org/2018/085>.

I am the sole author of [8]. I had a leading role in [5, 6, 7, 10]. I worked on the circuit design in [1], where the project was led by Jakub Breier. I analyzed the algorithm and security proof of ORANGE in [3]. In [9] I worked with Anupam Chattopadhyay, Shivam Bhasin and Jakub Breier on designing an automated ASIC design flow for LASER sensors in digital circuits. I worked on the design, analysis and hardware implementation of the countermeasure in [12]. I advised the main author of [11], Mahmoud A. Elmhohr, on hardware implementations and authenticated encryption and helped prepare the manuscript. I had a minor role in [2, 4] where I advised the main author on some of the issues related to my research and wrote some sections of the manuscripts.

06/01/2021

Mustafa Mahmoud Mohammed Kairallah

Acknowledgements

Firstly, I would like to express my sincere gratitude to my supervisor, Prof. Thomas Peyrin, for his continuous support of my research, his encouragement, motivation and knowledge. Prof. Peyrin has always helped me grow, presented me with opportunities to challenge myself and never held back on sharing his knowledge, experience or insights. He is great mentor and greater friend.

Secondly, I would like to thank my co-supervisor, Prof. Anupam Chattopadhyay, without whom I may not have had the opportunity to join the NTU community. He always provided me with career advices, insightful ideas and interesting research questions.

Besides my supervisors, I would like to thank the rest of my thesis advisory committee; Prof. Wu Hongjun and Prof. Khoo Khoongming, for their support and insightful comments throughout my PhD journey. I would, also like to thank my qualification exam committee, Prof. Fredrique Oggier and Prof. Wang Huaxiong, for their remarks and suggestions at the early stages of my PhD.

I am deeply thankful to Prof. Tetsu Iwata, Nagoya University, and Dr. Kazuhiko Minematsu, NEC, Japan, for their invaluable guidance, and for presenting me with the opportunity to learn from and work with two of the most important researchers in their field. I would like also to thanks Dr. Shivam Bhasin and Dr. Jakub Breier for their guidance on physical attacks and countermeasures in the early stages of my PhD.

I am grateful to Dr. Anne Canteaut and Dr. Gaëten Leurent and the rest of the SECRET Team in Inria, Paris for having me as a research visitor during the summer of 2019.

I am delighted to have shared the same lab, and a lot of delightful memories, with my colleagues; Wang Haoyang, Quan Quan Tan, Adrien Benamira, David Gerault, Anubhab Baksi, Zakaria Najm, Alexandre Adomanicai, Ling Sun, Sonu Kumar Jha, Vesselin Velichkov, Siang Meng Sim, Mohona Gosh, Ivica Nikolić, Zhenzhen Bao, Guozhen Liu, Sumit Kumar Pandey, Subhadeep Banik and Ling Song.

I am also grateful to all my co-authors from whom I got the chance to learn and grow: Thomas Peyrin, Anupam Chattopadhyay, Tetsu Iwata, Kazuhiko Minematsu, Shivam Bhasin, Jakub Breier, Zakaria Najm, Xialou Hou, Prassana Ravi, Sourav Sen Gupta, Anubhab Baksi, Karim M. Abdellatif, Rajat Sadukhan, Rajat Subhra Chakraborty, Debdeep Mukhopadhyay, Yang Liu, Bimal Mandal, Subhamoy Maitra, Mahmoud A. Elmoher, Sachin Kumar, Jawad Haj-Yahya, Vinay B. Y. Kumar, Suman Deb, Rupesh Kumar, Avi Mendelson, Yu Sasaki, Ling Sun, Siang Meng Sim and Radhamanjari Samanta.

I would like to thank my family for their continuous support. I would like to thank my friends who have supported me over the years during both my highest and lowest moments.

Finally, I would like to thank NTU and SPMS for the support throughout my PhD and for my NTU research scholarship. I would like to thank Temasek Labs @

NTU for their support of my research through funding some of my research trips.

Mustafa Khairallah
Singapore
06/01/2021

To the fond memory of my late father.

Contents

Statement of Originality	iii
Supervisor Declaration Statement	v
Authorship Attribution Statement	vii
Acknowledgements	xi
Table of Contents	xvi
Summary	xvii
I. Introduction	1
1. Motivation	3
1.1. Research Problem	4
1.2. Outline	5
2. Preliminaries and Background	7
2.1. Notation	7
2.2. Hardware Digital Circuit Design	8
2.3. Symmetric-Key Encryption	12
2.4. Block Ciphers	13
2.5. Hash Functions	19
2.6. Modes of Operation	22
2.7. Hardware Cryptanalysis	27
II. Hardware Implementation of Symmetric-Key Primitives and Attacks	35
3. On The Cost of ASIC Hardware Crackers	37
3.1. The Chosen-Prefix Collision Attack	38
3.2. Hardware Birthday Cluster	40
3.3. Hardware Differential Attack Cluster Design	42
3.4. Chip Design	43
3.5. Cost Analysis and Comparisons	51
3.6. Conclusion	59
4. Hardware Performance of the ΘCB3 Algorithm	61
4.1. Related Work	62
4.2. Proposed Architecture	62
4.3. Multi-Stream AES-like Ciphers	64

4.4. Implementations and Results	70
4.5. Conclusion	74
III. Tweakable Block Cipher Based-Cryptography	75
5. Arguments for Tweakable Block Cipher-Based Cryptography	77
5.1. History	77
5.2. The TWEAKEY Framework	78
5.3. TBC-Based Authenticated Encryption	80
5.4. Efficiency Function $e(\lambda)$	81
5.5. Applications and Discussions on the Efficiency Function	87
IV. TBC-Based Authenticated Encryption	89
6. Analysis of the Rekey-and-Chain Schemes	91
6.1. Background and Motivation	94
6.2. COFB-like Schemes	95
6.3. Application to COMET-128	98
6.4. Application to mixFeed	102
6.5. Misuse in RaC Schemes: Attack on mixFeed	107
7. Lightweight AEAD from Tweakable Block Ciphers	111
7.1. Specifications	113
7.2. Security Analysis	129
7.3. Design Rationale of Romulus and Remus	133
7.4. Hardware Implementations	135
7.5. Instantiation and Efficiency of Romulus and Remus	142
7.6. ASIC Design Space Exploration of Romulus-N1	145
7.7. Instantiation of Romulus and Remus with Skinny	148
V. Conclusions & Further Discussion	155
8. Conclusion	157
9. Other Works	159
10. Future Work	161
Bibliography	163

Summary

Lightweight cryptography is one the fastest growing areas in symmetric key cryptography. Its importance has increased due to applications such as pervasive computing and the Internet of Things (IoT). Among the goals of lightweight cryptography is authenticated encryption for constrained environments. Tweakable block ciphers have been proposed around two decades ago as new more powerful primitives compared to traditional block ciphers. Essentially, they provide simpler schemes that easier to understand and provide higher security bounds. Over the last few years, they have gained a growing attention due to their promising security properties. However, they have been less studied in the context of lightweight cryptography. It is in this spirit that we need to study the use of tweakable block ciphers for lightweight authenticated encryption, especially applications targeted towards hardware acceleration. We show that tweakable block ciphers offer very efficient schemes. They achieve competitive performance, very small area and strong provable security with large margins. In the first part of this thesis, we study some of the hardware implementation aspects of symmetric key cryptography. We study the implementation of cryptanalytic attacks with a case study on SHA-1, showing that attacks with time complexity $2^{64} \sim 2^{80}$ are practical, and discussing the different technologies that can be used to implement such attacks. We also study the hardware performance of the state-of-the-art tweakable block cipher-based mode ΘCB3 . We show that some properties that are good for software performance, *e.g.* parallelism, are not necessarily helpful in case of hardware implementation. Then, we analyze the potential of tweakable block cipher-based schemes. We show that for a given security level and under certain assumptions, they can offer the most efficient schemes in terms of the security-area-performance trade-off. Afterwards, we describe a framework for studying a class of tweakable block cipher-based schemes. We apply our framework to two of the round 2 candidates of the lightweight standardization project initiated by the National Institute of Standards and Technology (NIST). We show gaps in the initial analyses provided by the designers. Last but not least, we provide two families of tweakable block cipher-based schemes. The first family is **Romulus**. Its members enjoy full 128-bit AEAD security based on standard Tweakable Pseudo-Random Permutation (TPRP) assumptions. It has variants that are either nonce-respecting or nonce-misuse resistant. Its main distinguishing feature is that there is no state overhead beyond the tweakable block cipher's state. The second family is **Remus**. The tweakable block cipher used in **Remus** is built on top of a block cipher. It achieves flexible provable security between the birthday bound (64-bit security) and full 128-bit security based on the Ideal Cipher Model. The choice of the security level decides the state size, which ranges between 256 bits and 384 bits.

Part I.

Introduction

1. Motivation

Cryptography and hardware acceleration have always been entangled in a relationship that keeps driving both of them forward. This relation predates the invention of the transistor or the modern computer. The events surrounding World War II led to the emergence of modern cryptography. It changed cryptography from an art exclusive to the military and intelligence personnel to a scientific discipline practiced and studied by mathematicians and computer scientists, as well. Besides, it was discovered many years later that it was during World War II at Bletchley Park in England that the first automated hardware cryptanalytic machines were built. In 1939, Alan Turing designed a mechanical machine called "Bombe", which performed statistical attacks on German messages encrypted using Enigma [1] and intercepted by the British Navy. Between 1943 and 1945, British code breakers built another machine called "Colossus", which was used to break Lorenz, another German cipher. While Bombe was an electro-mechanical machine, Colossus used vacuum tubes and Boolean functions. It is considered the first programmable, electronic and digital computer. It is also described as the godfather of the modern computer [2].

Since then, modern computers have provided a rich platform to invent new cryptographic algorithms that are secure against simple cryptanalytic techniques used in the old days. Without modern computers, disciplines such as public-key cryptography and digital signatures may have not been possible. However, the invention of modern computers did not *only* kick-start new frontiers in the cryptographic research in terms of what applications or algorithms we can create. It has also led to research on how to make such algorithms secure, fast, efficient and both power- and energy-aware. With many competing design goals, a huge number of algorithms and implementation strategies have emerged over the years, each trying to target a certain application or computing environment. One approach to enhancing the efficiency of cryptographic algorithms is to merge different security goals in one algorithm in a way that is more efficient and robust than simply using two different algorithms.

One of the most popular applications of this approach is Authenticated Encryption (AE), where a private message M needs to be encrypted and at the same time the authenticity and integrity of M needs to be maintained. In a generalization of this problem, the message also includes a public part A , known as the associated data (AD), where the authenticity of both M and A together needs to be maintained. This scenario is known as Authenticated Encryption with Associated Data (AEAD) and is the main focus of this thesis.

Simultaneously, lightweight symmetric key cryptography has been a growing research area in the past 10 years or more. Lightweight cryptography is a wide umbrella that refers to algorithms designed with certain implementation metrics in mind, such as low area, low memory consumption, low power, or low energy. Its applications vary from block cipher (BC) design to AE or hash functions. This has led the National Institute of Standards and Technology (NIST) to release

1. Motivation

a call for proposals for a new lightweight cryptography standard [3]. The new standard is expected to be used in applications such as Internet-of-Things (IoT) and Sensor Networks. AEAD is one of the most important requirements of symmetric key cryptography in these environments since it is usually cheaper than having independent solutions for the authentication and encryption requirements of the system. Besides, generic combinations of encryption and authentication are not always secure. For example, Encrypt-and-MAC is a construction where the user independently applies encryption and authentication on the plaintext. If the scheme is not designed carefully, the MAC function may leak information about the plaintext. Most of the lightweight AEAD designs fall into one of three families: (Tweakable) Block Cipher-Based schemes, Permutation-Based schemes and Stream Cipher-Based schemes, with the last two families sometimes overlapping.

1.1. Research Problem

Many new ciphers, hash functions and operating modes have been recently proposed with “lightweightness” as the main target. CAESAR competition [4] for authenticated encryption has received many submissions aiming at lightweightness, and two schemes (ACORN [5] and Ascon [6]) were selected in the lightweight category. After several years of analysis and discussions, the NIST decided to organize a competition [3] to identify the future lightweight AEAD standard(s). One can separate the competition candidates into several classes: stream-ciphers, sponges, BCs and tweakable block ciphers (TBC).

Lightweight sponge-based constructions have been flourishing, as they can offer sufficient security with a small internal state. BC-based AEAD designs have been studied for a long time. AES-GCM [7] and OCB [8] are among the most famous BC-based modes. While they share the great advantage of being usable with widely deployed BC standards such as AES, most of them suffer from providing birthday-bound security, or low performance for beyond-birthday-bound security. For example, a BC-based scheme with birthday-bounded security requires a BC with n -bit block to achieve $\frac{n}{2}$ -bit security, while a scheme that can tolerate a smaller block size is secure beyond-birthday, and is usually harder/more costly to design. This is problematic for the lightweight cryptography scenario. For example, 64-bit lightweight BCs seem hardly usable in order to provide sufficient security. TBCs were introduced by Liskov *et al.* at CRYPTO 2002 [9]. Since their inception, TBCs have been acknowledged as a powerful primitive as they can be used to construct simple yet highly secure Nonce-based AEAD (NAE) or nonce Misuse-Resistant AEAD (MRAE) schemes, including Θ CB3 [10] and SCT [11]. Briefly, a nonce is an input value that is unique for each encryption call and is never repeated, while misuse resistance refers to schemes that can tolerate repetitions of the nonce. Indeed, TBC-based AEAD schemes such as Θ CB3 are very efficient in terms of the number of primitive calls and offer a high level of security as they achieve full n -bit security for a block size of n bits (in contrary to most BC-based modes). However, Θ CB3 uses an inverse function of the TBC, which means additional circuitry and a larger area. It also requires more storage than other modes, as it needs to keep track of the message check-sum and other internal values that are not part of the execution of the underlying TBC. These features are not suitable for lightweight devices. This leads to an important research problem:

Research Problem

Can we design AEAD from a TBC using standard security assumptions with Beyond-Birthday-Bound security, with no storage overhead beyond what is needed to compute the TBC?

While this is a loaded question we intend to dissect each of its elements throughout this thesis. In the first part, we will study how it is risky to rely on security levels of $64 \sim 80$ bits. We will also study how state-of-the-art TBC-based algorithms behave in hardware. Part II will explore the benefits of using a TBC. Then, we will introduce a framework for designing AEAD algorithms from TBCs, and we will show some examples of algorithms that fall within this framework. Finally, we give two algorithms that answer the aforementioned question elegantly. To sum up, the contributions in this thesis are three-fold:

1. Studying the state of the art hardware implementations of some of the aspects of symmetric-key cryptography, whether when it comes to cryptanalytic attacks or TBC-based AEAD algorithms.
2. Designing efficient algorithms of high performance and low cost, while keeping high security guarantees, helps the advancement of communication technologies in the era of the IoT and ubiquitous computing. The practical implications of the work described in this thesis are evident by its impact on the NIST lightweight cryptography standardization project [12], whether by the selection of the Romulus AEAD scheme for round 2 of the standardization process or by the impact of the attacks presented on the security claims and modifications of some of the competition candidates.
3. Investigating the capabilities of TBCs as tools to design AEAD: The results in this thesis show that TBCs are not *only* a great tool from a security point-of-view, but that they can also lead to very efficient implementations.

1.2. Outline

In this thesis, we focus on an intersection point where many of the design goals meet. This point presents security and efficiency challenges to cryptographic designers. While the title and chapters of this thesis delve into different cryptographic territories, *i.e.*, *design, analysis and implementation*, the common thread tying them together is *hardware-oriented AEAD*. We will start by introducing AEAD, presenting topics such as lightweight and hardware-oriented AEAD, discussing the NIST lightweight cryptography project, which has deeply impacted this thesis.

We will go on a journey through the spectrum of lightweight AEAD from TBCs. First, we study the hardware implementation of symmetric key cryptography through two aspects. We study the implementation of one of the most recent cryptanalytic attacks against SHA-1 in Chapter 3. In Chapter 4 we also study of the state-of-the-art TBC-based AEAD algorithm ΘCB3 . We show that while there is room for speeding up the implementation, the ΘCB3 algorithm cannot be considered a lightweight algorithm. We also show that when it comes to hardware,

1. Motivation

certain features that sound attractive in theory, *e.g.* parallelism, may not lead to the performance and efficiency gain expected.

Afterwards, we discuss some arguments in favor of using TBCs to design AEAD in Chapter 5. After that, we present the Rekey-and-Chain (RaC) framework in Chapter 6. We analyze some of the algorithms that happen to fall within this framework, showing potential design issues in these algorithms that show gaps in their security assumptions. These attacks have been followed by updated analyses by the designers capturing the issues we identified and follow up works further analyzing these schemes.

Given the insights presented in the previous chapters, we go on to design new AEAD algorithms that are meant to provide a wide range of security and efficiency trade-offs. This is presented in Chapter 7. The first family of algorithms, *Romulus*, offers the same security guarantees offered by ΘCB3 , both qualitatively and quantitatively. Yet, we forgo the parallelism offered by ΘCB3 in order to design a lightweight and very small TBC-based algorithm that requires very small overhead on top of the underlying TBC. In fact, some of the variants of *Romulus* offer even better security as they are secure even in scenarios of nonce misuse, at the expense of 50% performance overhead. The second family, *Remus*, offers a security trade-off, where we can develop very small implementations if lower security levels, either qualitatively or quantitatively, are tolerated.

While the thesis is written in this logical flow, the work in this thesis has followed a rather circular approach. For example, while a reader may consider that some of the attacks against *mixFeed* or *COMET-128* have impacted the design of *Romulus* or *Remus*, in some cases this has happened in reverse order. Since these four algorithms have been designed in parallel and independently, they all fall within the RaC framework, and some of the properties of *mixFeed* or *COMET-128* have actually been investigated while designing the preliminary versions of *Romulus* and *Remus*, and we use *mixFeed* or *COMET-128* as examples to highlight these properties. This does not only apply to security issues, but also performance and efficiency.

Before we start the aforementioned journey, it is important to highlight what expectations should the reader have besides the technical contributions. If a reader wants to leave this thesis with only one lesson learned, it is that hardware design and cryptography must go hand in hand, where the design of hardware-oriented cryptographic algorithms must be assisted by experimentation, measurements and practical implementations, as many assumptions on the performance that sound nice on paper usually fail in practice. At the same time, sacrificing qualitative security considerations for more efficient schemes can sometimes lead to significant security issues.

2. Preliminaries and Background

The Merriam-Webster dictionary [13] defines *cryptology* as

The scientific study of cryptography and cryptanalysis.

which tells us that cryptology, in its most primitive form, consists of two parts: hiding secrets, and unraveling those secrets. In fact, the progress of cryptology over the course of history has been pushed by a cat and mouse game between these two parts. Rulers, armies, and more recently internet users, try to hide sensitive information by using elaborate encryption methods, while their adversaries try to read this information. Initially, cryptography relied on making the information look like incomprehensible gibberish to unauthorized readers, while an authorized reader knows how to solve a puzzle, known as a cipher, that allows them to read the original information. Adversaries eventually manage to solve such puzzles, leading to the creation of more sophisticated ones. In the early days, the design and analysis of these ciphers was mainly the work of espionage agencies and military personnel and relied on paper-and-pen techniques and intuition. However, with the increasing complexity of designs and attacks, and the emergence of modern computers in the 20th century, cryptology became a scientific discipline practiced by mathematicians, computer scientists and engineers. Academic research has introduced new applications of cryptology, new cipher designs and new cryptanalysis techniques, and has often competed with the research coming out of intelligence agencies.

In the rest of this chapter, we explore some of the applications and concepts of modern cryptography, emphasizing some of the concepts discussed in this thesis. We provide some useful notations in Section 2.1. We discuss the hardware circuit design in Section 2.2. Then, we explain what is *Symmetric-Key Cryptography* (SKE) and the difference between SKE and *Public-Key Cryptography* (PKE) in Section 2.3. In Section 2.4, we discuss block ciphers (BC) and tweakable block ciphers (TBC), providing a brief description of the main features of 4 (T)BCs that we refer to throughout the thesis. In Section 2.5, we give a brief description for hash functions and SHA-1. We discuss encryption modes, authentication modes and authenticated encryption (AE) in Section 2.6. In Section 2.7, we provide a short survey on of hardware accelerators for cryptanalytic attacks, originally published in the *Proceedings of the 18th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS) 2018* [14].

2.1. Notation

Let $\{0, 1\}^*$ be the set of all finite bit strings, including the empty string ε . For $X \in \{0, 1\}^*$, let $|X|$ denote its bit length. Here $|\varepsilon| = 0$. For an integer $n \geq 0$, let $\{0, 1\}^n$ be the set of n -bit strings, and let $\{0, 1\}^{\leq n} = \bigcup_{i=0, \dots, n} \{0, 1\}^i$, where $\{0, 1\}^0 = \{\varepsilon\}$.

2. Preliminaries and Background

Let $\llbracket n \rrbracket = \{1, \dots, n\}$ and $\llbracket n \rrbracket_0 = \{0, 1, \dots, n-1\}$. Let $|X|_n = \max\{1, \lceil |X|/n \rceil\}$, where $n > 0$.

For two bit strings X and Y , $X \parallel Y$ is their concatenation. We also write this as XY if it is clear from the context. Let 0^i (1^i) be the string of i zero bits (i one bits), and for instance we write 10^i for $1 \parallel 0^i$. Bitwise XOR of two variables X and Y is denoted by $X \oplus Y$, where $|X| = |Y| = c$ for some positive integer c . For binary string X of $|X| \geq x$, we write $\text{lmt}_x(X)$ (resp. $\text{rmt}_x(X)$) to denote the leftmost (resp. rightmost) x bits of X . Note that we do not use “MSB” and “LSB” for them, which is customary but depends on endianness. Because our concrete specification (Section [7.7](#)) employs the little-endian format, our notation intends to avoid confusion. By convention, if one of X or Y is represented as an integer in $\llbracket 2^c \rrbracket_0$ we assume a standard integer-to-binary encoding, i.e., an integer $\sum_{i=0}^{n-1} x_i 2^i$ for $x_i \in \{0, 1\}$ is encoded to $(x_{n-1} \dots x_1 x_0) \in \{0, 1\}^n$. For example, $X \oplus 1$ denotes $X \oplus 0^{c-1}1$.

Parsing. For $X \in \{0, 1\}^*$, let $(X[1], \dots, X[x]) \stackrel{n}{\leftarrow} X$ be the parsing of X into n -bit blocks. Here, $X[1] \parallel X[2] \parallel \dots \parallel X[x] = X$ and $x = |X|_n$. $X[i]$ can interchangeably be written as X_i . Note that $|X[x]| < n$ if $|X|$ is not a multiple of n . When $X = \varepsilon$, we have $X[1] \stackrel{n}{\leftarrow} X$ and $X[1] = \varepsilon$. Note in particular that $|\varepsilon|_n = 1$.

Alternating Parsing. Let n and t be positive integers larger than 8. For $X \in \{0, 1\}^*$, let $(X[1], \dots, X[x]) \stackrel{n,t}{\leftarrow} X$ be the parsing of X into n -bit blocks and t -bit blocks in an alternating order. That is, we have $X[1] \parallel X[2] \parallel \dots \parallel X[x] = X$, where $|X[i]| = n$ for any odd $i \in \{1, \dots, x-1\}$, $|X[i]| = t$ for any even $i \in \{1, \dots, x-1\}$, $|X[x]| \in \llbracket n \rrbracket$ if x is odd, and $|X[x]| \in \llbracket t \rrbracket$ if x is even. When $X \neq \varepsilon$, x is determined as

$$x = \begin{cases} 2\lfloor |X|/(n+t) \rfloor & \text{if } |X| > 0 \text{ and } |X| \bmod (n+t) = 0 \\ 2\lfloor |X|/(n+t) \rfloor + 1 & \text{if } 1 \leq |X| \bmod (n+t) \leq n \\ 2\lfloor |X|/(n+t) \rfloor + 2 & \text{if } n < |X| \bmod (n+t) < n+t. \end{cases}$$

When $X = \varepsilon$, $X[1] \stackrel{n,t}{\leftarrow} X$ (thus $x = 1$) and $X[1] = \varepsilon$.

Galois Field. An element a in the Galois field $\text{GF}(2^n)$ will be interchangeably represented as an n -bit string $a_{n-1} \dots a_1 a_0$, a formal polynomial $a_{n-1}x^{n-1} + \dots + a_1x + a_0$, or an integer $\sum_{i=0}^{n-1} a_i 2^i$.

Matrix. Let G be an $n \times n$ binary matrix defined over $\text{GF}(2)$. For $X \in \{0, 1\}^n$, let $G(X)$ denote the matrix-vector multiplication over $\text{GF}(2)$, where X is interpreted as a column vector. We may write $G \cdot X$ instead of $G(X)$.

2.2. Hardware Digital Circuit Design

Digital circuits are circuits that operate on Boolean variables, where a variable can take two values: **True** or **False**, where usually **True** is represented as 1 and **False** is represented as 0. In practice, this can correspond to $+V_{dd}$ and 0 volts, respectively, where $+V_{dd}$ is the voltage of the power source of the circuit. These variables are operated on using Boolean Algebra, developed by George Boole in

1847 [15]. The operations of Boolean Algebra are the logical functions/gates: AND, OR, NOT, XOR, NAND, NOR, and XNOR. Given two Boolean variables x and y , a logic gate is given by its truth table, where the output is defined for every possible combination of inputs. In Table 2.1 we give three examples of such truth tables for the AND, XOR and NAND gates.

Table 2.1.: The truth table of the AND, XOR and NAND Boolean functions

x	y	AND : $x \wedge y$	XOR : $x \oplus y$	NAND : $x \nearrow y$
0	0	0	0	1
1	0	0	1	1
0	1	0	1	1
1	1	1	0	0

Bigger and more complex Boolean functions with larger number of variables can be built using smaller gates as their building blocks. Hence, a set of gates is called a universal set of gates if it can be used to implement any other Boolean function. One of the smallest universal set of gates is {NAND}, where any Boolean function can be built using only NAND gates. Another famous universal set of gates is {AND, XOR}. The symbols used to represent different gates are shown in Figure 2.1.

Figure 2.1.: The circuit symbols used for different gates.

Another important and widely used circuit component is called the multiplexer MUX. In its simplest form, a MUX has three inputs, two data inputs A and B and one control input X. If the value of X is 0, the output takes the value of A, and it takes the value of B, otherwise. The MUX symbol is shown in Figure 2.2.

Figure 2.2.: The circuit symbol of a multiplexer (MUX).

Using these primitives, two types of circuits can be built:

- *Combinational Circuits*: The circuit is built from a combination of gates,

2. Preliminaries and Background

implementing a given truth table. Given any set of inputs, the circuit outputs the required Boolean function.

- *Sequential Circuits:* For bigger and more complicated circuits, it is hard to implement the full circuit as a combinational one. Hence, the circuit is divided into smaller parts, where each part is implemented as a combinational circuit, and the parts are combined sequentially (iteratively). In order to implement such a circuit in hardware, we need to define additional components, namely, storage elements.

Figure 2.3.: The circuit symbol of a D-Flip-Flop (DFF).

There are different types of storage elements in practice. However, for the purposes of this thesis, we only focus on two types. The first type is the D-Flip-Flop (DFF), depicted in Figure 2.3. The DFF takes two inputs, a data input D , and a clock input CLK , and outputs Q . CLK is a periodic signal that oscillates between values 0 and 1, where the transition from 0 to 1 is called the *rising edge*. At every rising edge, the value of D is transferred to Q . Q maintains its value, otherwise. Since the DFF is more widely used, we simply refer to it as FF whenever there is no confusion. The second type of flip-flops we are interested in is the Scan-Flip-Flop (SFF), alternatively called Scan-Flop. The SFF behaves as a MUX and DFF combined in one circuit. Given a control input X , it selects between the two inputs D and S . It is depicted in Figure 2.4.

Figure 2.4.: The circuit symbol of a Scan-Flip-Flop (SFF).

Flip-flops can be used in sets of more than one to store multi-bit values. A set of n FFs is called an n -bit register. A simplified overall digital circuit is depicted in Figure 2.5. The circuit iterates over some combinational operations. The intermediate results stored in a register. A control sub-circuit is used to configure and control both parts.

In order to physically realize the circuit in hardware, there are two currently popular technologies:

Figure 2.5.: An overall sequential digital circuit.

- *Field Programmable Gate Arrays (FPGA)*: off-the-shelf devices that include configurable logic elements. These elements are called Look-Up Tables (LUT) and can be configured to perform different combinational logic circuits. They also include generic FFs that can be combined with the LUTs to construct sequential circuits. Usually, LUTs and FFs are grouped together in units called slices. FPGAs usually cost a few hundred dollars, but they are reusable and have a small engineering non-recurring cost, so they are suitable for testing new designs and for small scale projects.
- *Application-Specific Integrated Circuits (ASIC)*: built-to-order integrated circuits (IC) that perform a specific task. Usually, they are made in special fabrication facilities (*fabs*) and have a huge initial non-recurring cost, which makes them suitable for large scale projects. However, they are much cheaper than FPGAs once they are mass-produced. They also have better performance. In order to simplify the task of designing such circuits, the fab usually releases a so-called *standard cell technology library*, which is a set of primitive circuits that the designer can use to build his required IC. These libraries include all the logic gates with different performance parameters and some famous circuits as well, *e.g.* arithmetic addition circuits.

Once the circuit is realized in hardware, there are many different metrics that can be used to assess its cost and performance. In this section, we define only some of the main parameters, while other parameters are defined in different chapters if needed. These parameters are:

- *Area*: corresponds to the size of the circuit and the cost of realizing it. Theoretically, the area is measured in gate count, which is how many gates of each of the different primitive types we described are used in the circuit. For FPGA, the area is measured in terms of resource utilization, which can be in one of two flavors: either the amount of LUTs and FFs used, or the overall amount of slices. In practice, the amount of slices is what matters as it represents the overall physical area of the design. For ASIC, the area is measured in μm^2 , which is the physical circuit size. However, since this depends on the underlying technology library, it is usually normalized into *gate equivalents (GE)*. Usually, the NAND gate is the smallest two-input gate that can be realized in ASIC, and it also represents a universal set of gates

2. Preliminaries and Background

on its own. Hence, the area of a digital circuit is usually normalized as

$$\frac{\text{Area in } \mu m^2}{\text{Area of the smallest NAND gate in } \mu m^2}$$

where the area of the smallest NAND gate is denoted as 1 GE.

- *Throughput*: a measure of the speed of the circuit. It represents the amount of data that can be processed per second.
- *Latency*: the amount of time needed between receiving a set of inputs and producing the corresponding output.
- *Critical Path*: each gate consumes a small amount of time to produce its output called *gate delay*. For a set of gates that are connected serially, the overall additive delay between the input and output is called *path delay*. The *critical path delay* is the maximum path delay of the circuit. The clock period of the circuit cannot be smaller than its critical path. The critical path is measured for the combinational parts of the circuit, either from an input to an output, from an input to a register or from a register to an output.
- *Energy*: the amount of electrical energy required by the circuit to perform one full operation.
- *Power*: the rate at which electrical energy is consumed by the circuit while performing a certain task.

2.3. Symmetric-Key Encryption

Symmetric-Key Encryption (SKE) is a term that refers to a wide variety of cryptographic algorithms and technologies. It refers to scenarios where both the sender and receiver share the same secret key(s). In contrast, Public-Key Encryption (PKE) refers to algorithms where only one user knows the secret key, while other parties can only communicate with him/her using his/her public key. A serious issue related to PKE is the authentication of the public key, as the users need to be sure that the public key is indeed linked to Bob's identity. Hence, the public key has to be published over an authenticated channel. For example, in a public-key encryption scheme, Alice uses a public key shared by Bob to encrypt a message, where only Bob can decrypt such a message using his secret key. Bob's public and secret keys are mathematically linked together. In order for Bob to send a message to Alice, she would have to share her public key, as well. Basically, anyone who has Bob's public key can send him an encrypted message that only he can decrypt. On the other hand, in a symmetric-key encryption scheme, both Alice and Bob share the same key that can be used for both encryption and decryption. Only Alice and Bob can either encrypt or decrypt the data using that key, while an adversary, Eve, should only be able to see seemingly random gibberish. Such a communication channel is depicted in Figure [2.6](#).

PKE is more efficient in terms of the number of keys required to be communicated between multiple parties. A group of n users needs $O(n)$ keys to communicate securely, while for SKE you require $O(n^2)$ keys. On the other hand, in practice, SKE primitives are much more efficient than PKE primitives. Thus, what is

Figure 2.6.: Symmetric-key encryption

generally performed is that symmetric keys are first exchanged between two users with PKE techniques, then actual encryption/authentication of the data is performed using SKE.

SKE also refers to scenarios where the cryptographic algorithm is secret free, *e.g.* public hash functions or public permutations. In such scenarios, both Alice and Bob, as well as the adversary share the same level of information, and the security comes from certain assumptions on the public primitive. In general, SKE research can be summed up as the combination of the study of SKE primitives, *e.g.* (tweakable) block ciphers, stream ciphers and public permutations, and the study of the modes of operation of these primitives *e.g.* authenticated encryption and hash functions.

2.4. Block Ciphers

A block cipher (BC) is a keyed function $E : \mathcal{K} \times \mathcal{M} \rightarrow \mathcal{C}$, where $\mathcal{K} = \{0, 1\}^k$ is the key space, $\mathcal{M} = \{0, 1\}^n$ is the plaintext space and $\mathcal{C} = \{0, 1\}^n$ is the ciphertext space, *s.t.* for any $K \in \mathcal{K}$, $E(K, \cdot)$ is a permutation. $E(K, \cdot)$ can be interchangeably written as $E_K(\cdot)$. The diagrams of a BC is shown in Figures 2.7 and 2.8. The two diagrams can be used interchangeably. A secure BC should

Figure 2.7.: A block cipher $E_K(\cdot)$. Figure 2.8.: An alternative diagram for a block cipher $E_K(\cdot)$.

act as a Pseudo-Random Permutation (PRP), *i.e.*, it should satisfy that for a key

2. Preliminaries and Background

$K \xleftarrow{\$} \mathcal{K}$, $E_K(\cdot)$ is indistinguishable from a uniformly random permutation (URP) $P \xleftarrow{\$} \text{perm}(n)$, *i.e.*,

$$\text{Adv}_{\text{prp}}^E(q, \sigma) \stackrel{\text{def}}{=} \Pr[1 | K \xleftarrow{\$} \mathcal{K} : \mathcal{A}^{E_K(\cdot)}] - \Pr[1 | \mathcal{A}^{P(\cdot)} \xleftarrow{\$} \text{perm}(n)] \leq \alpha$$

s.t. α is negligible in n . This security model is the standard security model for block ciphers. Another useful security notion for block ciphers is the ideal cipher model (ICM), where a BC $IC : \mathcal{K} \times \mathcal{M} \rightarrow \mathcal{C}$ sampled uniformly at random from the set of all BCs with key space $\mathcal{K}$ and plaintext space $\mathcal{M}$ is an ideal cipher (IC). In such case, $IC_K(\cdot)$ is a random independent permutation over $\mathcal{M}$ for every choice of $K \in \mathcal{K}$.

A tweakable block cipher (TBC) is a keyed function $\tilde{E} : \mathcal{K} \times \mathcal{T}_W \times \mathcal{M} \rightarrow \mathcal{C}$, where $\mathcal{K} = \{0, 1\}^k$ is the key space, $\mathcal{T}_W = \{0, 1\}^t$ is the tweak space, $\mathcal{M} = \{0, 1\}^n$ is the plaintext space and $\mathcal{C} = \{0, 1\}^n$ is the ciphertext space, *s.t.* for any $K \in \mathcal{K}$ and $T_W \in \mathcal{T}_W$, $\tilde{E}(K, T_W, \cdot)$ is a permutation. $\tilde{E}(K, T_W, \cdot)$ can be interchangeably written as $\tilde{E}_K(T_W, \cdot)$ or $\tilde{E}_K^{T_W}(\cdot)$. The diagram of a TBC is shown in Figures 2.9 and 2.10. The two diagrams can be used interchangeably.

Figure 2.9.: A tweakable block cipher $\tilde{E}_K^{T_W}(\cdot)$.

Figure 2.10.: An alternative diagram for a tweakable block cipher $\tilde{E}_K^{T_W}(\cdot)$.

A secure TBC should act as a Tweakable Pseudo-Random Permutation (TPRP), *i.e.*, it should satisfy that for a key $K \xleftarrow{\$} \mathcal{K}$, $\tilde{E}_K^{T_W}(\cdot)$ is indistinguishable from a tweakable uniformly random permutation (TURP) $\tilde{P} \xleftarrow{\$} \text{perm}(n)$ for every choice of $T_W \in \mathcal{T}_W$, *i.e.*,

$$\text{Adv}_{\text{tprp}}^{\tilde{E}}(q, \sigma) \stackrel{\text{def}}{=} |\Pr[1 | K \xleftarrow{\$} \mathcal{K} : \mathcal{A}^{\tilde{E}_K^{T_W}(\cdot)}] - \Pr[1 | \mathcal{A}^{\tilde{P}(\cdot)} \xleftarrow{\$} \text{perm}(n)]| \leq \alpha$$

s.t. α is small.

There are different techniques for designing secure (T)BCs, *e.g.* Substitution-Permutation Networks (SPN) and Feistel Networks (FN). Throughout the thesis, we will not refer to the underlying structure of the cipher most of the time, and almost all the (T)BCs we will consider are SPNs. Hence, we give a brief description of SPNs and some related topics and leave other techniques outside the scope of this thesis.

The design principles of SPNs date back to 1949, when Claude Shannon proposed

building an iterative cipher using what he called diffusion and confusion [16]. The input is divided into small words. A non-linear operation is applied to each word independently (confusion). Then, a linear operation is applied over all the words (diffusion). Later, confusion was redefined as substitution, and diffusion was redefined as permutation. In order to build a cipher out of these two operations, they are repeated many times, where a key K_r is added after each iteration. One iteration is called an SPN round, while K_r is called a round subkey, or round key for short. One SPN round is depicted in Figure 2.11

Figure 2.11.: One round of an SPN.

The purpose of this structure is to resist known cryptanalytic attacks, such as linear and differential cryptanalysis. The input to each round is called a *state* and it consists of n bits. The n -bit state is divided into w words. The substitution layer (**SLayer**) performs confusion by applying a word-wise non-linear permutation, called the **SBox**. The **SLayer** may use identical or different **SBoxes** for each word. The goal of an **SBox** is to increase the complexity of the input/output relations. The state then is fed into a linear permutation (**PLayer**) which ensures that each output bit is affected by many input bits. By applying these two operations multiple times, followed by adding a key K_r each time, we ensure that the relations between the input block, round keys and output block are complex enough to resist statistical and cryptanalytic techniques. The round keys can be of large size, where for r rounds, we need $\Theta(rn)$ bits. In practice, the round keys are usually generated from a short secret key of size $\Theta(n)$ bits, using a *key scheduling* algorithm. Such an algorithm can be either non-linear, as in the case of AES [17], linear, as in the case of Skinny [18], or a simple repetition of the secret key, as in the case of LED [19].

In [20], Jean *et al.* introduced the tweakable framework, which can be used to design tweakable block ciphers from SPNs, by specifying a combined key scheduling algorithm that generates round keys as a function of a large key space $\mathcal{T}_K$, where $\mathcal{T}_K = \mathcal{T}_W \times \mathcal{K}$. The tweakable framework offers the flexibility to decide the parameters of the TBC, where the tweakable space can be arbitrarily divided into the secret key space and the public tweak space. This led to the development of simple, practical instantiation of TBCs and TBC-based modes of operation in the TPRP security model, such as Deoxys [21].

2. Preliminaries and Background

2.4.1. Hardware Implementations of SPNs.

Depending on the application and performance requirements, there are different architectures to implement the SPNs in hardware, whether it is targeted for ASIC or FPGA:

- *Unrolled implementation U*: An SPN is essentially just a digital circuit. It can be translated into a combinational circuit that processes all the inputs in one clock cycle. This architecture is characterised by a large critical path and area that increases with the number of rounds. However, in applications where the clock period is large and latency is more important, this architecture offers a good trade-off. It is particularly an attractive choice for FPGA implementations where there is usually unused logic on the device and the additional area required is not an issue.
- *Round-based implementation R1*: In order to reduce the critical path, area and take advantage of the iterative structure of SPNs, an R1 architecture is a sequential circuit which stores the state and current round key in a set of FFs and calculates one round per clock cycle at the expense of increased latency.
- *Round-based unrolled implementation Rx*: In order to achieve a better latency-critical path trade-off, we can implement the round-based architecture with many rounds per cycle.
- *Multi-stream implementation MSx*: both U and Rx can be adapted to operate on multiple blocks simultaneously, using the *pipelining* technique, where different parts of the SPN can be simultaneously operating on different blocks. This approach is particularly attractive for FPGA implementations (more on this in Chapter 4).
- *Word-serial implementation S*: In order to further reduce the area and amount of gates for critically low-area/power applications, we can build a high-latency, low-area implementation that operates on one byte or a small number of bytes and performs small operations, *e.g.* one SBox, per cycle. However, these implementations are usually very slow compared to the previous ones.
- *Bit-sliding implementation S1*: Jean *et al.* proposed this architecture at CHES 2017 [22] as a generalization for a long research line of serial implementations, *e.g.* [23, 24, 25]. The goal is to further reduce the area to extreme cases such as a single bit data-path.

Each of these architectures is suitable for different applications and trade-offs. Besides, understanding them can give new insights to the designers of BCs and modes of operations.

2.4.2. The Advanced Encryption Standard AES

The AES [26] is BC specification established by NIST in 2001 after a multi-year competition. It is a subset of the Rijndael BC family [17], designed by Daemen and Rijmen. It is probably the most studied/deployed cipher in the last 20 years.

It is built as an SPN, with a block size of 128 bits and $w = 8$. It has three variants, depending on the key size k and the number of rounds r , such that $(k, r) \in \{(128, 10), (192, 12), (256, 14)\}$. The state of the SPN is organized as a 4×4 byte matrix. A key scheduling algorithm is applied to the key to obtain $(r + 1)$ round keys, starting from K_0 up to K_r , each consists of 128 bits. K_0 is xored to the plaintext block. Each round, except the last round, consists of four sequential operations:

1. SubBytes: a non-linear Sbox is applied to each. The Sbox function is derived from the multiplicative inverse over $\text{GF}(2^8)$ and an invertible affine transformation.
2. ShiftRows: the bytes in each row of the state are cyclically shifted by a certain offset. The first row is unchanged while the second, third and fourth rows are shifted by one, two and three bytes, respectively.
3. MixColumn: Each column in the state is transformed using a linear operation. A constant matrix is left-multiplied by each column of the state.
4. AddRoundKey: For round i , K_i is xored to the state.

The final round skips the MixColumn operation, applying only SubBytes, ShiftRows and AddRoundKey. The AES key scheduling algorithm divides the input key into 32-bit words, and outputs an array of 32-bit words, where W_j is the j^{th} elements of the array and

$$K_{j/4} = W_j \parallel W_{j+1} \parallel W_{j+2} \parallel W_{j+3} \quad \forall 0 \leq j < 4r.$$

The algorithm uses a parameter $s = k/32$ that represents the number of 32-bit words in K . The first s words are given by

$$(W_0 \dots W_{s-1}) \stackrel{32}{\leftarrow} K,$$

while for $j \geq s$,

$$W_j = \begin{cases} W_{j-s} \oplus \text{SubWord}(W_{j-1} \ggg 8) \oplus \text{rcon}(j/s) & \text{if } j \equiv 0 \pmod{s} \\ W_{j-s} \oplus \text{SubWord}(W_{j-1}) & \text{if } s > 6, \text{ and } j \equiv 4 \pmod{s} \\ W_{j-s} \oplus W_{j-1} & \text{otherwise} \end{cases}$$

where $W \ggg l$ represents bitwise right rotation of W by l bits and $\text{rcon}(c)$ is defined as $x^{c \pmod{r}} \parallel 0^{24}$ such that x is defined over $\text{GF}(2^8)$. SubWord(W) applies the AES Sbox to each of the bytes of W , independently.

2.4.3. The Lightweight Encryption Device (LED) Cipher

LED is a lightweight BC proposed by Guo *et al.* [19] to target lightweight applications. It follows similar design principles to AES, with major differences. It has a 64-bit block. For simplicity, we describe two possible key sizes, 64 bits and 128 bits. The LED specification allows any key size in the range $64 \leq K \leq 128$. It operates on 4-bit nibbles instead of bytes, *i.e.*, $w = 4$. The 64-bit state is described as a 4×4 nibble matrix. One round of LED consists of four operations:

2. Preliminaries and Background

1. AddConstants: An encoding of the round index is added to the state of the cipher.
2. SubCells: The value of each nibble is replaced by another value based on the LED 4-bit Sbox.
3. ShiftRow: Each row of the state is cyclically shifted by a constant amount. The first row is left unchanged. The second, third and fourth rows are shifted by 1, 2 and 3 nibbles, respectively.
4. MixColumnSerial: A constant matrix is left-multiplied by each column, independently.

A notable feature of LED is that the round keys are not added to the state after each round. A sequence of 4 LED rounds is called a step. AddRoundKey is applied only to the input plaintext or after a full step. In the case of $k = 64$, the round keys are simply repetitions of K . In case $k = 128$, then two 64-bit subkeys are defined as $(K_1, K_2) \stackrel{64}{\leftarrow} K$ and the cipher alternates between them for odd and even steps.

2.4.4. Deoxys-BC

Deoxys-BC [21] is an ad-hoc TBC designed by Jean *et al.* based on the TWEAKEY framework. It is originally part of the Deoxys AEAD submission at the CAESAR competition, but it has been used as a stand alone TBC since then. It has a block size of 128 bits and a tweakkey size of either 256 or 384 bits, with 14 and 15 rounds respectively. The SPN is almost the same SPN as that of AES. Besides the different number of rounds, AddRoundKey is replaced with AddRoundTweakey and it also differs from AES in that all the rounds include the MixColumn step.

Another difference between AES and Deoxys-BC is the tweakkey scheduling algorithm. The tweakkey is divided into words of 128 bits. More precisely, if the tweakkey size is 256 bits it is divided into W_1 and W_2 , while if the tweakkey size is 384 bits it is divided into W_1 , W_2 and W_3 . The tweakkey-schedule state is initialized by

$$TK_0^1 = W_1, TK_0^2 = W_2, TK_0^3 = W_3$$

At round i , the round key is given by $TK_i^1 \oplus TK_i^2$ or $TK_i^1 \oplus TK_i^2 \oplus TK_i^3 \oplus RC_i$, where RC_i is a 128-bit round constant. The tweakkey-schedule state is updated as

$$TK_{i+1}^1 = h(TK_i^1), TK_{i+1}^2 = h(lfsr_2(TK_i^2)), TK_{i+1}^3 = h(lfsr_3(TK_i^3))$$

where $h(X)$ is a simple byte-reordering permutation on X and $lfsr_2(x)/lfsr_3(x)$ are two 8-bit Linear Feedback Shift Registers (LFSR) that can be implemented using one XOR gate per byte, each.

2.4.5. The SKINNY TBC

The Skinny family of TBCs [18] was proposed by Beierle *et al.* in Crypto 2016. The lightweight TBCs of the Skinny family have 64-bit and 128-bit block versions. However, we will only use the $n = 128$ bits versions in this thesis. The internal state is viewed as a 4×4 square array of cells, where each cell is a byte. We denote $IS_{i,j}$ the cell of the internal state located at Row i and Column j (counting

starting from 0). One can also view this 4×4 square array of cells as a vector of cells by concatenating the rows. Thus, we denote with a single subscript IS_i the cell of the internal state located at Position i in this vector (counting starting from 0) and we have that $IS_{i,j} = IS_{4 \cdot i + j}$.

Skinny follows the TWEAKEY framework from [20] and thus takes a tweakkey input instead of a key or a pair key/tweak. We denote $z = k/n$ the tweakkey size to block size ratio. The tweakkey state is also viewed as a collection of z 4×4 square arrays of cells. We denote these arrays $TK1$ when $z = 1$, $TK1$ and $TK2$ when $z = 2$, and $TK1$, $TK2$ and $TK3$ when $z = 3$. Moreover, we denote $TK_{z,i,j}$ the cell of the tweakkey state located at Row i and Column j of the z^{th} cell array. As for the internal state, we extend this notation to a vector view with a single subscript: $TK1_i$, $TK2_i$ and $TK3_i$.

The cipher receives a plaintext $m = m_0 \| m_1 \| \dots \| m_{14} \| m_{15}$, where the m_i are bytes. The initialization of the cipher's internal state is performed by simply setting $IS_i = m_i$ for $0 \leq i \leq 15$:

$$IS = \begin{bmatrix} m_0 & m_1 & m_2 & m_3 \\ m_4 & m_5 & m_6 & m_7 \\ m_8 & m_9 & m_{10} & m_{11} \\ m_{12} & m_{13} & m_{14} & m_{15} \end{bmatrix}$$

The cipher receives a tweakkey input $tk = tk_0 \| tk_1 \| \dots \| tk_{30} \| tk_{16z-1}$, where the tk_i are 8-bit cells. The initialization of the cipher's tweakkey state is performed by simply setting for $0 \leq i \leq 15$: $TK1_i = tk_i$ and $TK2_i = tk_{16+i}$ when $z = 2$, and finally $TK1_i = tk_i$, $TK2_i = tk_{16+i}$ and $TK3_i = tk_{32+i}$ when $z = 3$.

The Round Function.

Skinny has 40, 48 and 56 rounds, for $z = 1, 2$ and 3 , respectively. One encryption round is composed of five operations in the following order: SubCells, AddConstants, AddRoundTweakey, ShiftRows and MixColumns. The operations are similar to the operations of AES with a few major differences:

1. SubCells applies the same lightweight 8-bit Sbox to every byte of the state, independently. This Sbox can be implemented with only 8 XOR gates and 8 NAND gates.
2. AddRoundTweakey adds a 64-bit round tweakkey to half the state only in order to reduce the number of XOR gates required.
3. MixColumn uses a lightweight constant matrix has only 0 and 1 coefficients.

The tweakkey scheduling algorithm is similar to that of Deoxys-BC with a different permutation. It generates only 64-bit tweakkeys.

2.5. Hash Functions

Cryptographic hash functions are one of the main and most widely used primitives in symmetric-key cryptography. They have different features and requirements compared to (T)BCs). One of their key applications is to provide data integrity by ensuring each message will lead to a seemingly random digital *fingerprint*.

2. Preliminaries and Background

They are also used as building blocks of some digital signature and authentication schemes. A cryptographic hash function takes a message of arbitrary length as input and returns a fixed-size string, which is called the hash value/tag. In order for the function to be considered secure, it must be hard to find collisions, i.e. two or more different messages that have the same tag. More specifically, a n -bit cryptographically secure hash function that generates an n -bit tag must satisfy at least the security notion of collision resistance, i.e. finding a pair (M_1, M_2) of distinct messages, such that $H(M_1) = H(M_2)$ must require about $2^{n/2}$ computations.

2.5.1. The Davies-Meyer Construction.

The Davies-Meyer (DM) construction shown in Figure 2.12 is a method to convert an ideal cipher to a collision-resistant fixed-length compression function. The function takes an $(n + k)$ -bit input and returns n -bit output, satisfying the collision resistance requirement.

Figure 2.12.: The Davis-Meyer construction

2.5.2. The Merkle-Damgård Construction.

Given the collision-resistant compression function $T = h(X)$ that compresses n bits into t bits, the Merkle-Damgård (MD) construction [27] is a method to convert this function into a hash function of arbitrary input length. The input message M is divided into k -bit blocks M_i , where k is the key size. The last block encodes the length of the message. The compression function is applied iteratively, such that $T_0 = IV$ is a constant and $T_i = h(M_i \| T_{i-1})$. The hash function output is T_m , where $m = \lceil \frac{|M|}{n-t} \rceil$. An two-block example is shown in Figure 2.13, where the compression function is based on the DM construction.

Given its specific structure, the MD construction may be vulnerable to collision attacks that take advantage of its properties. Two example of these attacks are the identical-prefix collision attack and the chosen-prefix collision attack:

1. *Identical-Prefix Attack (IPC)*: Instead for searching for a random pair of colliding messages, the adversary can search for two messages that are identical up to the i^{th} block. He/she can first generate T_i and search for a pair of block-sequences $(M_{i+1} \| \dots \| M_{i+c}, M'_{i+1} \| \dots \| M'_{i+c})$, such that $T_{i+j} = T'_{i+j}$ and there is at least one index j where $M_{i+j} \neq M'_{i+j}$.
2. *Chosen-Prefix Attack (CPC)*: The adversary can choose two different prefixes and generate a pair (T_i, T'_i) , such that $T_i \neq T'_i$. Then, he/she searches for

Figure 2.13.: The Merkle-Damgård construction

a pair of block-sequences $(M_{i+1} \parallel \dots \parallel M_{i+c}, M'_{i+1} \parallel \dots \parallel M'_{i+c})$, such that $T_{i+j} = T'_{i+j}$. This attack is more powerful as it offers more freedom to the adversary. The attacker can choose any two arbitrary messages and then append them with different suffixes that lead to a collision. It was proposed by Stevens *et al.* in [28].

2.5.3. SHA-1 and Related Attacks.

The SHA-1 hash function defines a generalized-Feistel-based compression function used inside the Merkle-Damgård (MD) algorithm. It was selected in 1995 as a replacement for the SHA-0 hash function after some weaknesses have been discovered in the latter. While the two functions are relatively similar, SHA-1 was considered collision resistant till 2005, when Wang *et al.* proposed the first cryptanalytic attack on SHA-1 [29]. Since then, a lot of efforts have been targeted towards making the attack more efficient. In 2015, the authors of [30] provided an estimation for finding near collisions on SHA-1, which is a critical step in the collision attacks. The authors provided a design of an Application-Specific Instruction-set Processor (ASIP), named Cracken, which executes specific parts of the attack. It was estimated that to execute the free-start collision and real collision attacks from [31], the attacks will take 46 and 65 days and cost 15 and 121 Million Euros respectively. At Eurocrypt 2019, Leurent and Peyrin [32] provided a chosen-prefix attacks which uses two parts: first a birthday search to reach an acceptable set of differences in the chaining variable, and then a differential cryptanalysis part that successively generate near-collision blocks to eventually reach the final collision. The attack was implemented on GPUs and a first chosen-prefix collision was published in January 2020 [33].

2.5.4. Birthday Search in Practice.

An adversary searching randomly for a collision on a hash function requires $O(2^{n/2})$ executions of the hash function, on average, in order to find a collision. This is known as the birthday search problem. The efficient design of a collision search algorithm is not a trivial task, especially if the attacker wants to use parallelization over a set of computing machines. The challenges arise in memory consumption and comparison operations. This issue is discussed in details in [34]. Assume the hash function operates only on messages of fixed length, then we define a graph based $h(x)$, where there is a directed edge from x to $h(x)$. The collision search

2. Preliminaries and Background

problem can be treated as a graph search problem, where the attacker is looking for two edges with the same endpoint but with different starting points. Pollard's rho method [35] helps finding a collision in the functional graph with a small memory requirement. The underlying idea is to start at any vertex and perform a random walk in the graph until a cycle is found. Unless the attacker is unlucky to have chosen a starting point that is part of the cycle, he ends up with a graph that resembles the Greek letter ρ and the collision is detected. Unfortunately, this method is not efficiently parallelizable, as it provides only $\mathcal{O}(\sqrt{m})$ speed-up when m cores are used. In [34], the authors proposed a method to achieve $\mathcal{O}(m)$ speed-up, using limited memory and communication requirements. This algorithm leads to very efficient parallel implementations, and is the basis for our study in Chapter 3.

2.6. Modes of Operation

An SKE primitive such as a BC is a useful tool, but on its own, it has limited functionality. For example, a naive approach to use a BC to encrypt messages of arbitrary size, also known as the Electronic Code Book mode (ECB), is to divide the message into blocks of equal size and sequentially apply the BC to each of these blocks. Figure 2.14 shows an image encrypted using ECB. Even though we cannot exactly deduce everything about the image without the secret key, the adversary can obviously observe that it includes a mountain, just by looking at the encrypted image. The original image can be seen in Figure 2.15, where it is clear that encryption, in this case, did not offer the required security. Hence, the ECB mode in most cases does not offer the required confidentiality for the encrypted messages. The main cause of this issue is that the ECB mode

Figure 2.14.: An image encrypted using the ECB mode [36]

does not hide non-random patterns in the plaintext, where two equal blocks of the plaintext get encrypted to two equal blocks of the ciphertext. Hence, in order to maintain privacy, the ciphertext is required to look random. We can define the privacy requirement of an encryption mode as the indistinguishability of the ciphertext generated using that mode from a uniformly random string.

Figure 2.15.: The original image of Figure 2.14 before encryption [37]

In that case, the indistinguishability is defined as the indistinguishability of the encryption oracle from a random oracle, *i.e.*, an oracle that outputs a uniformly random string independent from the input. One famous mode of operation that satisfies this requirement is the Ciphertext-Block-Chaining (CBC) mode, shown in Figure 2.16. Given a random initial vector IV , and a plaintext that consists of multiple blocks, the output ciphertext block for a given plaintext block is added to the next plaintext block before calling the BC.

Figure 2.16.: The CBC mode

While privacy is a critical security requirement, it is not enough to ensure safe communications, as modes like CBC do not guarantee the authenticity of the ciphertext. For example, an adversary can change the ciphertext, or replace the ciphertext string with a random string, and the receiver would normally decrypt this malicious string without issues. The malicious ciphertext can be complete gibberish or it can be a small modification in the original ciphertext, *e.g.* change a bank transaction from 100 USD to 1000 USD. In either case, the implications can be devastating. Hence, another popular security goal is authentication, where an authentication mode is designed such that the receiver can verify whether the received string is generated by the legitimate sender or not, and reject illegitimate messages, generating an invalid flag denoted as $\perp$. Such modes are called Message Authentication Codes (MAC) in the SKE setting. In MACs, the adversary tries to generate/modify strings such that one of these malicious messages eventually gets

2. Preliminaries and Background

accepted.

Most applications that use SKE require both privacy and authentication. However, combining two modes; one that achieves privacy and one that achieves authentication, is both expensive and not always secure. For example, an encrypt-and-MAC plaintext approach refers to the application of two independent schemes on the plaintext; one for encryption and one for authentication. It is clearly insecure as an adversary can tamper the output of the encryption scheme without changing the MAC output. At the same time, it costs at least the sum of the cost of the two schemes. More examples of insecure constructions can be found in [38]. In [38], Bellare and Namprempre introduced the concept of authenticated encryption, which specifies modes that achieve both privacy and authentication simultaneously and potentially at a lower cost. In some scenarios, the message includes public information that does not need to be encrypted but needs to be authenticated as part of the message. To cover such scenarios, Rogaway [39] extended this notion to Authenticated Encryption with Associated Data (AEAD). Over the past 20 years, a lot of research has been done in this domain, crowned by the conclusion of the CAESAR competition [4]. In the CAESAR competition, many teams from all over the world worked on building and analyzing AEAD schemes, targeting different use cases; lightweight, high-performance and defense-in-depth. Over 5 years, 57 designs were analyzed over 4 rounds, and a portfolio of 4 winners and 2 second-choice algorithms covering these use cases was eventually selected.

2.6.1. The Security Notions of AEAD

An AEAD scheme Π consists of two algorithms: the encryption algorithm

$$\Pi.\mathcal{E} : \mathcal{K} \times \mathcal{N} \times \mathcal{A} \times \mathcal{M} \rightarrow \mathcal{C} \times \mathcal{T}$$

and the decryption algorithm

$$\Pi.\mathcal{D} : \mathcal{K} \times \mathcal{N} \times \mathcal{A} \times \mathcal{C} \times \mathcal{T} \rightarrow \mathcal{M} \cup \perp$$

where $\mathcal{K} = \{0, 1\}^k$ is the key space, $\mathcal{N} = \{0, 1\}^{nl}$ is the nonce space, $\mathcal{M} = \{0, 1\}^*$ is the message space, $\mathcal{C} = \{0, 1\}^*$ is the ciphertext space, $\mathcal{T} = \{0, 1\}^\tau$ is the tag space and $\perp$ represents an invalid message string. For simplicity, we use $\Pi.\mathcal{E}_K^N(\cdot, \cdot)$ and $\Pi.\mathcal{D}_K^N(\cdot, \cdot, \cdot)$ to represent the encryption and decryption algorithms for a key K and a nonce N . An AEAD scheme Π usually relies on an underlying cryptographic primitive Π_{prim} : *e.g.* BC, IC or an ideal permutation (IP). An AEAD Oracle $\mathcal{O}_\Pi$ accepts two types of queries from an adversary $\mathcal{A}$: encryption queries and decryption queries. In the real world, $\mathcal{O}_\Pi$ calls $\Pi.\mathcal{E}_K^N(\cdot, \cdot)$ for encryption queries and $\Pi.\mathcal{D}_K^N(\cdot, \cdot, \cdot)$ for decryption queries. In the ideal world, $\mathcal{O}_\Pi$ calls a random oracle RO for encryption queries and outputs $\perp$ for decryption queries. An RO outputs a uniformly random string of the same length as the input message M in addition to the tag length, *i.e.*, $|M| + \tau$, *s.t.* for the same input N, A, M , it always outputs the same output C, T . For the sake of the analysis in this thesis, we combine the ideal world encryption and decryption parts into one oracle. There are two main notions of AEAD security that we are interested in this thesis:

- *Nonce-Based Authenticated Encryption (NAE)* [40, 41, 42] In this model, the value of the nonce N is unique for each query, *i.e.*, the adversary is not allowed to repeat N for two different messages.

- *Nonce-Misuse-Resistant Authenticated Encryption (MRAE)* In this model, the adversary is allowed to repeat/reuse N for multiple different queries.

Security Notions for NAE. We consider the standard security notions for nonce-based AE [40, 42, 43]. Let Π denote an NAE scheme consisting of an encryption procedure $\Pi.\mathcal{E}_K^N(\cdot, \cdot)$ and a decryption procedure $\Pi.\mathcal{D}_K^N(\cdot, \cdot, \cdot)$, for secret key K chosen uniform in the set $\mathcal{K}$ (denoted as $K \xleftarrow{\$} \mathcal{K}$). For a plaintext M with a nonce N and associated data A , $\Pi.\mathcal{E}_K^N(\cdot, \cdot)$ takes (N, A, M) and returns ciphertext C (typically $|C| = |M|$) and tag T . For decryption, $\Pi.\mathcal{D}_K^N(\cdot, \cdot, \cdot)$ takes (N, A, C, T) and returns a decrypted plaintext M if the authentication check is successful, and otherwise an error symbol, $\perp$.

The privacy notion is the indistinguishability of encryption oracle $\Pi.\mathcal{E}_K^N(\cdot, \cdot)$ from the random-bit oracle $\$$ which returns random $|M| + \tau$ bits for any query (N, A, M) . The adversary is assumed to be nonce-respecting (NR), i.e., nonces can be arbitrarily chosen but must be distinct for encryption queries, while they can be repeated for decryption queries. We define the (NR) privacy advantage as

$$\text{Adv}_{\Pi}^{\text{priv}}(\mathcal{A}) \stackrel{\text{def}}{=} \Pr \left[K \xleftarrow{\$} \mathcal{K} : \mathcal{A}^{\Pi.\mathcal{E}_K(\cdot, \cdot, \cdot)} \Rightarrow 1 \right] - \Pr \left[\mathcal{A}^{\$(\cdot, \cdot, \cdot)} \Rightarrow 1 \right],$$

which measures the hardness of breaking the privacy notion for $\mathcal{A}$.

The authenticity notion is the probability of successful forgery via queries to $\Pi.\mathcal{E}_K$ and $\Pi.\mathcal{D}_K$ oracles. We define the (NR) authenticity advantage as

$$\text{Adv}_{\Pi}^{\text{auth}}(\mathcal{A}) \stackrel{\text{def}}{=} \Pr \left[K \xleftarrow{\$} \mathcal{K} : \mathcal{A}^{\Pi.\mathcal{E}_K(\cdot, \cdot, \cdot), \Pi.\mathcal{D}_K(\cdot, \cdot, \cdot)} \text{ forges } \right],$$

where $\mathcal{A}$ forges if it receives a value $M' \neq \perp$ from $\Pi.\mathcal{D}_K$. Here, to prevent trivial wins, if $(C, T) \leftarrow \Pi.\mathcal{E}_K(N, A, M)$ is obtained earlier, $\mathcal{A}$ cannot query (N, A, C, T) to $\Pi.\mathcal{D}_K$. The adversary is assumed to be nonce-respecting for encryption queries.

Security Notions for MRAE. We adopt the security notions of MRAE following the same security definitions as above, with the exception that the adversary can now repeat nonces in both encryption queries. Such an adversary is called nonce-misusing (NM)¹. We write the NM-privacy advantage as

$$\text{Adv}_{\Pi}^{\text{nm-priv}}(\mathcal{A}) \stackrel{\text{def}}{=} \Pr \left[K \xleftarrow{\$} \mathcal{K} : \mathcal{A}^{\Pi.\mathcal{E}_K(\cdot, \cdot, \cdot)} \Rightarrow 1 \right] - \Pr \left[\mathcal{A}^{\$(\cdot, \cdot, \cdot)} \Rightarrow 1 \right],$$

and the NM-authenticity advantage as

$$\text{Adv}_{\Pi}^{\text{nm-auth}}(\mathcal{A}) \stackrel{\text{def}}{=} \Pr \left[K \xleftarrow{\$} \mathcal{K} : \mathcal{A}^{\Pi.\mathcal{E}_K(\cdot, \cdot, \cdot), \Pi.\mathcal{D}_K(\cdot, \cdot, \cdot)} \text{ forges } \right].$$

We note that while NM adversaries can repeat nonces, without loss of generality, we assume that they do not repeat the same query [44].

2.6.2. The Θ CB3 AEAD mode

The Θ CB3 AEAD mode was proposed in 2011 by Krovetz and Rogaway [10] as a tool to analyze the the OCB BC-Based mode. It is one of the early modes based on an ad-hoc TBC, and it was instantiated in the CAESAR

¹It may also be called Nonce Repeating or Nonce Ignoring.

2. Preliminaries and Background

competition [4] by Jean *et al.* as Deoxys-I [21], with the Deoxys-BC adhoc TBC. The mode demonstrates the advantages of TBCs in terms of beyond-birthday-bound n -bit security and very simple security proofs.

Figure 2.17.: The encryption section of Θ CB3

Figure 2.18.: The PMAC authentication section of Θ CB3

The Θ CB3 AEAD mode consists of two parts, shown in Figures 2.17 and 2.18. We assume the message M consists of m blocks and AD consists of a blocks. The TBC-based PMAC mode in Figure 2.18 is applied to AD , where the tag is stored in an auxiliary storage. The message is encrypted as shown in Figure 2.17. Independently, an XOR check-sum of the blocks of M is accumulated. Finally, the check-sum is encrypted using the TBC and its encryption is XORed to the tag of AD to generate the overall tag. The design requires that all the tweak inputs of the TBC, *i.e.*, T' , T_i and T_j for $0 \leq i < m$ and $0 \leq j < a$, are pairwise distinct. In other words, each TBC call must have a unique tweak. It is a nonce-respecting mode and its security is based the TPRP assumption. Recently, it was also instantiated as SKINNY-AEAD [45] using the Skinny TBC, and is currently a second-round candidate for the NIST lightweight standardization project.

2.6.3. The combined-feedback (COFB) AEAD mode

The COmbined FeedBack mode (COFB) [46] is a popular lightweight BC-based AEAD mode that was proposed by Chakraborti *et al.* at CHES 2017. It is the basis of the GIFT-COFB NIST candidate [47]. It is based on the PRP security of the underlying BC. Given a message of m blocks and AD of a blocks, and assuming the input consists of full blocks, the COFB mode is shown in Figure 2.19.

The output of the first BC call is truncated to $\frac{n}{2}$ bits and used as a mask for the subsequent BC inputs. After each cipher call, it is multiplied by a constant over $\text{GF}(2^{64})$. The ρ is a linear invertible transformation that operates on $2n$ bits. It is used to update the internal state of the algorithm and generate the ciphertext block simultaneously. The COFB mode offers a good trade-off for lightweight applications as it requires a state of only $1.5n + k$ bits. On the flip side, the security is limited to only $\frac{n}{2} - \log(n)$ bits. Similar to OCB, the first stage of its analysis is based on an idealized TBC variant called iCOFB

Figure 2.19.: The COFB AEAD mode

2.7. Hardware Cryptanalysis

Hardware cryptanalysis has always been an important part of modern cryptography. It studies building application-specific electronic machines for performing cryptanalytic attacks. These machines can use different technologies, starting from mechanical computers during World War II, to FPGA, GPU or ASIC in the modern days. A widely held belief is that FPGAs and GPUs are suited for small-scale or low-budget computations, while ASIC is predicted to be better for heavy computational tasks or if the attacker has a huge budget to spend. It is intuitive that a chip that is designed for a specific task is much more efficient than a general-purpose chip for the same task. However, since ASIC design has a huge non-recurring cost for fabrication, it is only competitive when a huge amount of chips is required. Besides, unlike the cryptographic algorithms themselves, which are usually optimized for hardware implementations, the cryptanalytic algorithms are usually designed for general-purpose computing machines. Hence, it is not necessarily true that ASIC implementations of such algorithms are more efficient. In other words, ASIC can always be at least as efficient as general-purpose CPUs or GPUs, as in the worst case the ASIC designer can simply design a circuit that is similar to the general-purpose one, but the gap in efficiency between the ASIC and the general-purpose circuit depends on the algorithm being implemented.

2. Preliminaries and Background

In general, ASIC provides an unfair advantage to players with bigger budgets. This has led to speculation that large intelligence entities may already possess ASIC hardware crackers that can break some of the widely used cryptographic schemes. In this chapter, we address the question of the feasibility of such machines and whether it is more beneficial to use ASIC for cryptanalysis. The answer to this question is yes, but only for generic attacks of very large complexities, e.g. $> 2^{64}$. For low scale or more complicated cryptanalytic attacks, GPUs provide a very competitive option, due to reusability, mass production and/or the possibility of renting them.

A relevant topic to our study is blockchain mining. As discussed earlier, big players can gain a huge advantage by using expensive ASICs. This has been a trend for Bitcoin specifically, where the introduction of a new ASIC machine lowers the profitability of older machines significantly. To maintain fairness of blockchain and cryptocurrency mining, memory-bound and ASIC-resistant hashing algorithms have been used, such as Ethash [48] for the Ethereum cryptocurrency and the X16R algorithm [49].

2.7.1. Cryptanalytic Attacks with Tight Hardware Requirements

Most of the cryptanalytic attacks on ciphers, especially symmetric-key ciphers (block ciphers, stream ciphers, hash functions, etc) consist of at least one phase of executing a complex search algorithm. With the huge input and output spaces of the involved functions, this step is very expensive in terms of time and/or memory consumption. Hence, acceleration algorithms and machines usually target efficient and parallelizable implementations of this step. In [50], the authors provided three assumptions that cover a wide variety of attacks and help develop efficient accelerators:

1. Cryptanalytic algorithms are parallelizable.
2. Different computational nodes need to communicate with each other only for a very limited amount of time.
3. Since the target algorithms are computationally intensive, the communication with the host is very limited compared to the time spent on the computational tasks.

In this section, we describe how some of the costly attacks can be adjusted in order to satisfy these conditions and lead to efficient hardware accelerators. For a wider exploration of different cryptanalytic techniques, we refer the interested reader to any of these resources: [51, 52, 53].

2.7.2. Brute-Force Attacks

Brute-Force attacks play an important role in the security of ciphers, especially in the field of symmetric-key cryptography. Brute-force attacks refer to attacks where all the possible values of a secret variable are tried until the correct value (or a set of valid values) is reached. This type of attack is applicable to any cryptosystem and provides an upper bound on the computational complexity of breaking a cipher. For example, since the Data Encryption Standard (DES) uses a secret key of 56 bits, it requires at most 2^{56} encryptions/decryptions in order to significantly

narrow down the space of possible secret keys. Hence, it was believed when DES was introduced, that it has a security level of 56 bits. Any attack that requires less than 2^{56} encryptions/decryptions is considered a genuine threat to the security of the cipher. Moreover, while 2^{56} operations was considered beyond the realm of possibility in the 1980s and early 1990s, the NIST organization has recently announced that any security level below 112 bits will be considered insecure from now on [3]. However, implementing brute force attacks is not as straightforward as it sounds and its practicality is not just subject to the availability of resources. A lot of challenges face the attackers when it comes to memory management, parallelization and data sorting/searching.

2.7.3. Time-Memory-Data Trade-off Attacks

In order to overcome the high time complexity required by most attacks, there is a trade-off to be made between the time and space requirements. For example, considering a block cipher $E_K(X)$, which represents a family of bijective permutations parametrized by the key value K , the attacker can choose on one end of the trade-off to ask for $E_K(0)$ and use brute-force to try all possible keys until he finds the correct key (or set of keys, as depending on the size of X and K it may not be possible to find a unique solution using only a single encryption), or the attacker can pre-compute and sort $E_{K'}(0) \forall K' \in 2^k$, then for any instance of the block cipher the key recovery takes $\mathcal{O}(1)$ as it involves only one memory access. However, the space complexity of the later case is $\mathcal{O}(2^k)$. TMDT attacks try to find a sweet spot between these two extremes, making complex attacks more practical for implementations. In this section, we describe two of the famous examples for such attacks.

Meet-in-the-Middle Attacks The Meet-in-the-Middle (MitM) attack was introduced by Diffie and Hellman in 1977 [54]. It applies to scenarios where an intermediate value during a function execution can be represented as the output of two independent random mappings. The most famous example for a cipher vulnerable to this attack is 2DES, which consists of applying the Data Encryption Standard (DES) twice using two independent keys, i.e.,

$$C = E_{K_2}(E_{K_1}(P))$$

A straightforward brute force attack would require 2^{112} operations, since DES has a key size of 56 bits. However, the MitM attack requires only 2^{57} operations and works as follows. First, we notice that

$$I = E_{K_1}(P) = E_{K_2}^{-1}(C)$$

Second, we notice that I is a collision between two random mappings. Hence, finding a pair (K_1, K_2) , such that $E_{K_1}(P) = E_{K_2}^{-1}(C)$, falls under the birthday problem and requires only $2^{\frac{|K_1|+|K_2|}{2}}$, i.e., 2^{56} iterations, on average. Since, every iteration consists of one encryption and one decryption operations, the overall number of function calls is 2^{57} , which is only twice the brute force complexity against DES. On the other hand, as most birthday attacks, the MitM requires a huge memory space, since every execution has to be stored and compared to all previous executions until the collision is found.

2. Preliminaries and Background

Hellman Time-Space Trade-off This attack, first proposed by Hellman in 1980 [55], targets accelerating brute force search. For simplicity, we consider only the case where the key size equals the plaintext size. However, the same attack can be generalized to other cases. The attacker chooses a plain-text P , which he knows will be encrypted at some point in the future. Then, he selects a set of possible keys as the starting point of his computation. For example, he can use the set $\{K_i | K_i \in [0, N]\}$, where N is one of the parameters of the time-space trade-off. The next step is to compute $C_i^0 = E_{K_i}(P)$. The attacker iterates over the computation $C_i^j = E_{C_i^{j-1}}(P)$, where $j \in [1, S]$ (S is the second parameter of the trade-off). All the previous steps are done offline, without communicating with the target user. At this point, the attacker ends up with N chains, each has $S + 1$ nodes, as follows:

$$\begin{aligned} C_0^0 &\rightarrow C_0^1 \rightarrow C_0^2 \dots \rightarrow C_0^S \\ C_1^0 &\rightarrow C_1^1 \rightarrow C_1^2 \dots \rightarrow C_1^S \\ &\vdots \\ C_N^0 &\rightarrow C_N^1 \rightarrow C_N^2 \dots \rightarrow C_N^S \end{aligned}$$

The previous lists include $(S + 1) * (N + 1)$ Keys. In order to save memory, the attacker stores only the initial key K_i and the final value C_i^S . Next, in the online phase, the attacker intercepts a ciphertext $C = E_{K^*}(P)$. First, he compares C to the $N + 1$ final values in his pre-computed table. If $C = C_i^S$, since $C = E_{K^*}(P)$, then $E_{K^*}(P) = E_{C_i^{S-1}}(P)$. The attacker returns $K^* = C_i^{S-1}$. Otherwise, the attacker sets $C^0 = C$ and computes the list in the same manner as the lists generated during the offline phase:

$$C^0 \rightarrow C^1 \rightarrow C^2 \dots \rightarrow C^S$$

If $C^l = C_i^S$, then $E_{K^*}(P) = E_{C_i^{S-l-1}}(P)$. In both cases, the attacker needs to recompute the chain where the collision occurred until the key value is found. On average, the online phase requires $S/2$ operations to find the collision and $S/2$ operations to find the key. However, since the attack covers only $(S + 1) * (N + 1)$ key candidates, it has a success probability of $\frac{(S+1)*(N+1)}{|\mathcal{KS}|}$, where $|\mathcal{KS}|$ is the size of the full key space.

2.7.4. Parallel Birthday Search Algorithms

So far, we have described three different generic attacks against ciphers. All these attacks have one feature in common. A cipher is considered as a random mapping $E : \mathcal{K} \times \mathcal{M} \rightarrow \mathcal{C}$. In this section, we consider a wider class of functions; collision-resistant compression functions $T = H(x) : \{0, 1\}^n \rightarrow \{0, 1\}^t$, such that $n \gg t$ and it is computationally hard to find a pair (x_1, x_2) such that $H(x_1) = H(x_2)$. This class of functions has several applications in the construction of secure hash functions and the cryptanalysis of symmetric key ciphers. For examples, the problem of finding such a collision is helpful to the meet in the middle attack described earlier. It is known that the computational complexity of finding a collision for such a function is upper bounded by the birthday bound $2^{t/2}$. However,

Figure 2.20.: A simplified functional graph example. An edge goes from vertex a to vertex $H'(a)$

the efficient design of a collision search algorithm is not a trivial task, specially if the attacker wants to make use of parallelisation over a set of computing machines. This issue is discussed in details in [34]. First, we look at the problem of designing an efficient algorithm for the birthday search problem on a single processing unit. A straightforward approach is to compute $2^{t/2}$ random instances. With high probability, a colliding pair exists in the list formed by these instances. However, such approach requires $\mathcal{O}(2^{t/2})$ memory locations and $\mathcal{O}(2^t)$ memory accesses/comparisons. In order to overcome these tight requirements, several attacks with different trade-offs have been proposed, almost all of them share the same property; the function in question is reduced to $T = H'(x) : \{0, 1\}^t \rightarrow \{0, 1\}^t$, which is treated as a pseudo-random function (PRF). One of the useful ways to represent such a function is using a functional graph, which is a directed graph with 2^t vertices and two vertices x and y with an edge from x to y are connected if $y = H(x)$, as shown in Figure 2.20

The collision search problem can be treated as a graph search problem, where the attacker is looking for two edges with the same endpoint but with different start-points. Pollard's rho method [35] helps find a collision in the functional graph with a small memory requirement. The underlying idea is to start at any vertex and perform a random walk in the graph until a cycle is found. Unless the attacker is unlucky to have chosen a starting point that is part of the cycle, he ends up with a graph that resembles the Greek letter ρ and the collision is detected.

Nonetheless, Pollard's rho method cannot be efficiently parallelized without modifications. If an attacker tries to run many instances of the algorithms on several machines, independently, each machine will try to look for a cycle in a specific part of the functional graph. However, there is no guarantee that the first colliding pair will be found using a single machine, as each member of the pair can be found using a different machine. For example, two machines can enter the same cycle, but the attacker will not detect this event. Hence, if he uses m machines, he will need $\mathcal{O}(\frac{2^t}{\sqrt{m}})$ time to find the collision, as opposed to his original target of $\mathcal{O}(\frac{2^t}{m})$. This is still better than $\mathcal{O}(2^t)$, but it is sub-optimal as it can be improved to $\mathcal{O}(\frac{2^t}{m})$.

In [34], the authors proposed a method to achieve $\mathcal{O}(\frac{2^t}{m})$ speed-up, using limited memory and communication requirements. First, the attacker defines a distinguished point to be a point $H'(x)$ that has a special property which can be easily checked, e.g. the first d bits are equal to 0. Second, the attacker chooses m random messages $x_0^1, x_0^2, \dots, x_0^m$ and assigns one of them to each machine. Third, each machine i computes a trace $(x_0^i, x_1^i, \dots, x_d^i)$, where $x_j^i = H'(x_{j-1}^i)$ and x_d^i is a distinguished point. This is a random walk in the functional graph. If the

2. Preliminaries and Background

Figure 2.21.: An example of two colliding traces in the functional graph

probability of the condition $x_d^i = H'(x_{d-1}^i)$ is θ , the average length of the trace d is $\frac{1}{\theta}$. Hence, if θ is too large, the traces are too short and the total number of traces $\approx \frac{2^k}{\theta}$ is in the same order of magnitude of 2^k . This means that the attacker does not observe a significant reduction in the memory usage compared to a random-search based algorithm. On the other hand, if θ is too small, the traces become so long, and there is a risk of hitting a cycle within the trace, without ever hitting a distinguished point. Hence the choice of θ is crucial for the attack efficiency. Moreover, it is a good practice to abort the trace after it becomes too long, e.g. $\frac{20}{\theta}$. The previous two steps are repeated $\frac{2^{t/2\theta}}{m}$ times. Each trace is specified in the memory as (x_0^i, d, x_d^i) . Finally, a central server needs to sort these traces efficiently, according to the values x_d^i and find the traces that have the same endpoint. Once two similar traces are found, it is easy to find a collision within them that looks like in Figure 2.21.

2.7.5. Hardware Machines for Breaking Ciphers

Brute Force Machines

In this section we describe cases where engineers have been able to build machines to efficiently execute brute force attacks against certain ciphers.

Deep Crack In 1998, the Electronic Frontier Foundation (EFF) built a dedicated hardware machine consisting of 1856 ASIC chips connected to a single PC. The machine was able to test over 90 billions DES keys per second, which means that it can go over all the 2^{56} DES keys in 9 days. It was able to solve one of the RSA security DES challenges in 56 hours. The project costed 250,000 US Dollars and was motivated by the discrepancy between the estimates of academia and government officials regarding the cost and time required to break DES [56].

COPACOBANA In CHES 2006, COPACOBANA [50] was introduced as an FPGA cluster architecture consisting on 120 FPGAs controlled by a host computer. It is considered to be the first publicly reported configurable platform built specifically for cryptanalysis as its main purpose. The design philosophy behind the architecture depends on the three main assumptions in Section 2.7.1. These assumptions are satisfied by both brute force and cryptanalytic attacks. Hence, the COPACOBANA has been used to accelerate several attacks [57]. We sum up some of these attacks in Table 2.2. Some of the attacks performed on the COPACOBANA platform, e.g. guess and determine attack on A5/1, were specially designed in the first place to make use of hardware acceleration [58].

WindsorGreen In 2016, a document was release accidentally on the New York University server, describing a custom made supercomputer designed by the NSA

Table 2.2.: Some of the attacks performed using the COPACOBANA platform

Cipher	Attack Type	Time Consumed
DES	Brute Force	6.4 days
A5/1	Guess and Determine [58]	6 hours
ECC (k=79)	Discrete-Log	3.06 hours
ECC (k=97)	Discrete-Log	93.4 days

and IBM, which is believed to have mainly two applications, cracking ciphers and forging cryptographic signatures [59]. However, limited information is available publicly on the project.

The Factoring Machine

The problem of efficiently finding the prime factors of an integer is one of the oldest mathematical problems in the field of Number Theory. It is also the basis of some of the Public Key Cryptosystems (PKC), such as RSA. If a computer can factor $n = pq$ into p and q , it would lead to breaking RSA systems of key size $|n|$. The Number Field Sieve (NFS) algorithm is one of the famous algorithms for solving the factoring problem. However, efficient and cheap implementations of this algorithm are non-trivial. In [60], the author describes three different architectures for machine to perform the NFS algorithm, the cheapest of which costs 400,000 US Dollars and is estimated to take under one year to break RSA-768 in under one year. However, the results are highly speculative and are not supported by any actual implementation.

Blockchain Mining

While the topic of blockchain mining is not directly related to cryptanalysis or breaking ciphers (specifically hash functions), it is closely related to the acceleration of brute force attacks. The mining operation involves finding an input block to a secure hash function such that the output tag is less than a specific value, i.e. has a certain number of leading 0's. The number of required leading 0's defines the complexity of the problem, which is equivalent to a pre-image attack against a truncated version of the hash function. If that version of the hash function is pre-image resistant, then the mining step is equivalent to a brute force pre-image attack. As the blockchain gets older, the mining step gets more complex. Hence, several industrial players have been interested into accelerating these computations. In 2016, Intel applied for a patent for a Bitcoin mining hardware accelerator [61], which consists of a processor and a coupled hardware accelerator that uses SHA-256 as the main underlying hash function. It is claimed that this system can reduce the power consumption involved in Bitcoin mining by 35%. In April 2018, Samsung has also confirmed that it is building ASIC chips to mine Bitcoin. The new chips utilize the technology and expertise of Samsung's high memory capacity GPUs and can be designed for a specific hash function, to give customers the freedom to choose the target blockchain, not being limited to Bitcoin. However, once fabricated the chip is hash-function specific. It is supposed to increase the power efficiency by 30% and to execute 16 Tera hashes per second [62].

Part II.

Hardware Implementation of Symmetric-Key Primitives and Attacks

3. On The Cost of ASIC Hardware Crackers

This chapter we discuss the hardware implementation of a group of cryptanalytic attacks against SKE primitives. As a case study, we consider attacks against one of the recently broken primitives, SHA-1. It is an attempt at answering three important research questions:

- *Can the cost of the collision attacks against SHA-1 be reduced?* There has been major breakthroughs in the cryptanalysis of SHA-1 over the past few years, with the first practical identical-prefix collision (IPC) found in February 2017 [63] and the first chosen-prefix collision (CPC) found in January 2020 [33]. While these attacks are practical on general-purpose GPUs, they still take a few months to generate one collision, by both academic and industrial entities. Interestingly, the authors of [33] remarked that TLS and SSH connections using SHA-1 signatures to authenticate the handshake could be attacked with the SLOTH attack [64] if the chosen-prefix collision can be generated quickly. Hence, we would like to check if ASIC can provide a better alternative to speed up the attacks, using larger budgets. We actually show that chosen-prefix collisions could be generated within a day or even a minute using an ASIC cluster costing a few dozen Million USD (the amortized cost per chosen-prefix collision is then much lower).
- *What is the difference between generic search-based attacks and cryptanalytic attacks in terms of cost and implementation?* When analyzing a new cipher, any algorithm that has a theoretical time and data complexities lower than the generic attacks is considered a successful attack and the cipher is considered broken. For example, an n -bit hash function that is collision resistant up to the birthday bound is considered insecure if there is a cryptanalytic attack that requires less than $2^{0.9n/2}$ hash calls. Most of the time, researchers only measure time complexity in terms of function calls and ignore other operations required to perform the attack if they are much smaller. However, in practice, it can be a lot harder to implement a cryptanalytic attack compared to a generic attack, even with lower theoretical complexity. There are countless attacks published every year with a complexity very close to the generic one, but a natural example of such scenarios is the biclique attack against AES [65], where the brute force complexity is reduced only by a small factor from 2^{128} to $2^{126.1}$. However, one can question if implementing the simple brute force attack would actually be much less complex in practice. In this chapter, we compare the generic 64-bit birthday CPC attack over a 128-bit hash function to the cryptanalytic CPC attack against SHA-1 (which costs close to $2^{63.6}$ operations on GPUs, and of a lower complexity in theory) showing that in practice, the generic attack cost is more than 5 times cheaper than the ad-hoc CPC attack. Attacks like biclique or complex cryptanalysis

3. On The Cost of ASIC Hardware Crackers

are even more difficult to implement than the ad-hoc CPC attack and might require a huge memory, which probably makes the gap even larger. Hence, we argue that for a cryptanalytic attack to be competitive against a generic algorithm in practice, one must ensure a sufficiently large gap, at least of a factor 5, if not more (only an actual hardware implementation testing or estimation could give accurate bounds on that factor).

- *How secure is an 80-bit collision-resistant hash function?* In the NIST Lightweight Cryptography Workshop 2019, Tom Brostöm proposed an application for lightweight cryptography where the SIMON cipher [66] is used in the Davis-Meyer construction as a secure compression function which is collision-resistant at most up to 2^{64} computations [67]. Besides, it remains a common belief that SHA-1 is insecure due to the cryptanalytic attacks against it, but it would have still been acceptable otherwise. Actually, it is only since 2011 that 80-bit security is not recommended anymore by the NIST, and 80-bit security for data already encrypted with this level of protection is deemed acceptable as a legacy feature, accepting some inherent risk. Hence, we study the cost of implementing the generic 2^{80} birthday collision attack against SHA-1, showing that it is within our reach in the near future, costing ≈ 61 million USD to implement the attack in 1 month, which is not out of reach of large budget players, *e.g.* large government entities, and with the decreasing cost of ASICs, this will even be within reach of academic/industrial entities in the near future.

Finally, we argue that ASIC provides the most efficient technology for implementing high complexity and generic attacks, while GPU provides a competitive option for cryptanalytic and medium/low cost attacks.

3.1. The Chosen-Prefix Collision Attack

The chosen-prefix collision attack against SHA-1, it is not applied directly to the compression function of SHA-1, but to a helper function, as the input and output sizes need to be equal. Let IV_i represent a chaining value to the compression function (reached after processing a prefix), x a message block, and $H(IV_i, x)$ the application of the SHA-1 compression function. The goal of the birthday phase of CPC attack is to find many solutions x_1 and x_2 such that $L(H(IV_1, x_1)) = L(H(IV_2, x_2))$, where $L(x)$ is a linear function applied to a word x , in order to select some of the output bits of the compression function. The helper function is defined as:

$$f(x) = \begin{cases} L(H(IV_1, x)), & \text{if } x = 1 \pmod{2} \\ L(H(IV_2, x)), & \text{otherwise.} \end{cases} \quad (3.1)$$

When a collision $f(x_1) = f(x_2)$ is found, we have $x_1 \neq x_2 \pmod{2}$ with probability one half, and in this case we obtain $L(H(IV_1, x_1)) = L(H(IV_2, x_2))$.

3.1.1. Differential Cryptanalysis.

In this section, we briefly describe the algorithms involved in the second part of the chosen-prefix collision attack: the generation of successive near-collision blocks to reach the final collision. The details of this differential attack can be found

3.1. The Chosen-Prefix Collision Attack

in [29, 31, 68, 69, 63, 32, 33]. For each new near-collision block, the attacker has to go through three main steps:

1. Preparing a fully defined differential path for the SHA-1 compression function (in particular a non-linear part has to be generated for the first few steps of the SHA-1 compression function)
2. Find base solutions for the first few steps of this differential path (a base solution is simply two messages inputs that verify the planned differential path in the internal state up to the starting step of the neutral bits).
3. Expand those solutions into many solutions using what is known as neutral bits (in order to amortize the cost of the base solution), and check whether any of these solutions verify the differential path until the output of the compression function.

A neutral bit for a step i is a bit (or a combination of bits) of the message such that when its value is flipped on a base solution valid until step i , the differential path is still satisfied with high probability until step i . Most of the time, a neutral bit is a single bit, but it can sometimes be composed of a combination of bits. A neutral bit for a step i allows to amortize the cost of finding a solution to the differential path until step i .

The hardware cluster we consider consists of one master node and many slave nodes. The master builds a proper differential path for the compression function steps, based on the incoming chaining values, and generates base solutions based on this path. The slave is then required to expand these base solutions into a wider set of potential solutions and find out which of them satisfy the differential path until a certain step r (we selected $r = 40$ for ASIC for implementation efficiency purposes, but we remark that $r = 61$ was selected for GPU even though it does not have much impact) in the SHA-1 compression function. The master then aggregates all the solutions that are valid up to step r and exhaustively search for solutions that are valid up to step 80. This is repeated several times until a valid solution for the differential path is found. Consequently, we define a slave as a dedicated core that is responsible for extending a base solution found by the master into a set of potential solutions by traversing the tree of solutions defined by the neutral bits.

Unfortunately, this attack is not hardware-friendly and needs a lot of control logic. The master has to send to the slave:

1. A base solution, which consists of two message blocks M_1 and M_2 .
2. A set $[DP]$ of differential specifications for the slave to check conformance.
3. A group of neutral bit sets N_i , where the neutral bits in N_i are supposed to be neutral up to step i .

Combining a base solution (M_1, M_2) valid at step i and the set N_i , we get about $2^{|N_i|}$ new solutions that are valid up to step i , simply by trying all the possible combinations of the neutral bits in the set. In a naive approach, each of these partial solutions is expended to $2^{|N_{i+1}|}$ by applying combinations of the next set. Eventually, we would end up with $2^{\sum_i |N_i|}$ partial solutions, organised in a tree as shown in Figure 3.1. However, the neutral bits N_{i+1} are defined such that they

3. On The Cost of ASIC Hardware Crackers

don't impact the path up to step $i + 1$. Therefore, if the partial solution does *not* satisfy the conditions at step $i + 1$, there is no need to apply the neutral bits N_{i+1} , and we can instead cut the corresponding branch from the tree. Indeed, there is a certain probability that a solution valid at step i will be valid at step $i + 1$, according to the SHA-1 differential path selected. With the parameters used in SHA-1 collision attacks, most subtrees fail.

We can generate the partial solutions using a graph search algorithm to start navigating the tree from its root, and neglect complete subtrees that are failing. In our study, we choose Depth-First Search (DFS) graph search, with some modifications to suit our specific problems, in order to satisfy our assumptions for the cryptanalytic algorithm, as DFS has low memory requirements.

Figure 3.1.: Building partial solutions with neutral bits

Our attack scenarios. We consider three attack scenarios:

1. A plain 2^{64} birthday search: a generic birthday attack against a 128-bit hash function, constructed by selecting only 128 bits out of the 160 output bits of the SHA-1 compression function.
2. A plain 2^{80} birthday search: a generic birthday search over the full space of the SHA-1 compression function.
3. The chosen-prefix collision attack on SHA-1 from Leurent and Peyrin [32, 33].

These three scenarios cover two generic attacks against two security levels used in practice and one cryptanalytic attack.

3.2. Hardware Birthday Cluster

In this section, we describe the hardware core that handles the birthday attack. First, we define the nodes used in the proposed cluster. Then, we describe the design of the slave nodes and the communication requirements.

3.2.1. Cluster Nodes

The cluster used to apply the parallel birthday search attack consists of two types of nodes:

1. *Master*: a software-based CPU that manages the attack from high level and performs some jobs including choosing the initial prefixes, distributing the attack loads among slaves, sorting of the outputs and identifying colliding traces.
2. *Birthday Slaves*: dedicated cores that can perform different parts of the parallel birthday search. Specifically, it compute traces in the functional graph of the function in question, and once the master has identified colliding traces, the core also can locate the exact collision in these traces.

3.2.2. Hardware Design of Birthday Slaves

The design of the proposed birthday slave is shown in Figure 3.2. It's main role is to iterate the helper function of Equation 3.1. It consists of a reconfigurable ROM, where the initial trace value x_0 , IV_1 and IV_2 are loaded, a logic SHA-1 core which performs the step function of SHA-1, a comparator to compute $L(x)$, $x \pmod{2}$ and check whether a given x is a distinguished point (see [34]) or not, a memory to store distinguished points and a control unit to handle the communications with the master, and measure the lengths of different traces.

Figure 3.2.: Birthday slave for the parallel collision algorithm

In order to estimate the cost of the proposed core, the area and speed are compared to a single, step-based SHA-1 core, which is a standard practice in estimating the cost of SHA-1 cryptanalytic attacks. We have implemented a full SHA-1 core and it has an area of 6.2 KGE and 0.21 ns critical path. The implementation of the core in Figure 3.2 using a step-based SHA-1 core requires at least twice this area. Moreover, its critical path is dominated by the memory and counters in the control logic. Besides, it is not expected that a huge ASIC cluster will run at a speed higher than 1 GHz, due to the power consumption. Hence, in order to regain the efficiency lost due to the extra control logic and memories, it is a good approach to try to use this logic with as many SHA-1 steps as possible. Given these experiments and the huge cost of the control overhead, we increase the efficiency by cascading 4 SHA-1 steps instead of one in the SHA-1 core. This makes the critical path around 1ns, but a full SHA-1 computations takes only 20 cycles instead of 80, and the overhead 25% instead of 100%.

3.3. Hardware Differential Attack Cluster Design

In this section we discuss the challenges and different trade-offs when implementing the neutral bit search algorithm in ASIC and give a description of the circuit. The cluster architecture uses 3 types of nodes: master nodes, birthday slaves (BC), and neutral bit slaves (NB).

3.3.1. Neutral Bits

One of the trade-offs when implementing the attack is whether to consider neutral bits as only single-bits or to use the more general sets of multi-bits. The first approach leads to a very small circuit, but it strongly limits the number of usable neutral bits. This increases the overall work load, as more base solutions need to be generated, and more time is spent applying neutral bits. The second approach is more complex, because multi-bit neutral bits must be represented by a bit-vector. However, the single-bit neutral bits are not sufficient to implement an efficient attack, and we have to use the second option:

1. Our simulation results show that the success probability of single-bits is very low. Hence, any gain achieved by using them is offset due to the huge work load and high communication cost between the master and slave.
2. In order to achieve significant results, multi-bits are inevitable. In particular, *boomerangs* [70] (which can be seen as multi-bit neutral bits with extra conditions to reach a later step) are crucial cryptanalytic tools for a low-complexity attack against SHA-1. Hence, avoiding multi-bits can lead to a drastic loss in terms of attack efficiency.

3.3.2. Storage

Each multi-bit neutral bit is represented by a 512-bit vector, which indicates the location of the involved bits in the message block (a SHA-1 message block is indeed 512-bit long). However, we noticed that almost all the neutral bits involve bits only in the last 6 32-bit words of the message block. Therefore, we reduced the representation to only 192 bits. Yet, since the original chosen-prefix collision attack against SHA-1 uses ~ 60 neutral bits, including boomerangs, this requires a representation of $\sim 11,520$ bits. Besides, the last few levels of the tree requires 320 bits per neutral bits as the boomerangs can be located as early as step 6. In addition, for each level of the tree search we need a counter to trace which node we are testing. The tree used in the attack has ~ 10 levels, and our experiments show that the maximum number of neutral bits in one level is ~ 26 bits. Hence, the overall size of the counters is ~ 260 bits. In order to design the circuit that handles this tree search algorithm, we tried out four different approaches:

1. Generic approach: we assume that each tree level can have ~ 28 neutral bits (slightly higher than our experiments for tolerance). Also, assume that these levels can be related to any step of the SHA-1 compression function between 10 and 26, i.e. 16 possible steps. In total, this requires $\sim 63,670$ memory locations (Flip-Flops).
2. Statistical approach: from the software experiments and simulations, we identified an average number of neutral bits per level. In the design, we use

the maximum number of neutral bits we observe for each level (in addition to two extra bits for tolerance). We observed that only the first few levels require such a huge storage, while the later levels usually have $3 \sim 7$ bits per level. In addition, boomerangs are usually $3 \sim 4$ per level. This reduces the memory requirement by about 50%. However, it remains a huge requirement.

3. Configurable approach: our experiments showed that not only the number of neutral bits per level can be predicted, but also the values of these bits. In other words, very few bits have different values for different blocks. Hence, we can fix each neutral bit to two or three choices and use flip-flops to configure which choice is selected during execution. This reduces the cost significantly. However, the cost is still high as a multiplexer has an area only $\sim 50\%$ of a flip-flop. Besides, we still need flip-flops for configuring these multiplexers.
4. Another approach is to reduce the cost by fixing the the neutral bit values to a set of statistically dominant values. Indeed, [33] reports using the same neutral bits for each near-collision block. This eliminates the need to store the neutral-bit reference values.

At the end, we chose the third approach, since our analysis shows that it captures the reality, while allowing some level of freedom for the attacker to adjust the attack parameters after fabrication.

3.3.3. Architecture

Figure 3.3 shows the architecture of the neutral-bit slave. It consists of a register file to store the differential path for comparison, a configurable ROM to store the base solution, a unit to enumerate the different neutral bit patterns and maintains the tree level for the graph search algorithm, and the SHA-1 step logic.

Figure 3.3.: Neutral-bit slave hardware architecture

3.4. Chip Design

In this section, we describe our process for simulating the proposed chips and the results in terms of power, area and performance for each.

3.4.1. Chip Architecture

A challenge when designing this cluster is the communication overhead between the master and the slaves. A 100MHz SPI bus interface is used as a one-to-one

3. On The Cost of ASIC Hardware Crackers

communication interface with the attack server. A set of ASICs can also be daisy-chained, thanks to this interface, in such a way to lower the number of interconnects with the master. It provides enough bandwidth to handle the data exchanges between the BD/NB slave cluster and the attack server. The Control Unit (CU) is responsible for dispatching the 32-bit de-serialized packets sent by the attack sever to configure the BD/NB slaves. It is also responsible for daisy-chaining and demultiplexing the output traces of the different BD/NB slaves to the SPI bus interface before the serialization. Each ASIC also outputs an asynchronous interrupt signal. The interrupt signal is 1 when at least one BD/NB slave is done, and an output trace is available. Those interrupt signals will be managed by the PYNQ board cluster interfaces. Table 3.1 provides the I/O list of the overall chip consisting of multiple slaves.

Figure 3.4.: System architecture of the ASIC cluster chip

Table 3.1.: I/O for Proposed Chip

Pin	Direction	Description
n_reset	input	asynchronous system zero reset
sysclk	input	internal logic clock signal
spiclk	input	SPI interface logic clock signal
mosi	input	internal logic clock signal
ss	input	SPI slave select signal
miso	output	internal logic clock signal
interrupt	output	global alarm output signal

3.4.2. Implementation

Early studies [71] demonstrated the effectiveness of body biasing in reducing leakage, improving performances, and worst case power consumption. This is an interesting feature for high performance computing, and practical cryptanalysis. Indeed this feature allows to get the best possible performance at a given desired energy point. For a single targeted attack, the energy cost is not the critical factor in the overall attack cost. However it has a direct impact on the complexity of the cooling infrastructure when the attack complexity gets high. Moreover, for multiple attacks scenario, the energy becomes a critical factor.

The STMicroelectronics CMOS FD-SOI 28nm technology has been chosen for our simulations for its very good *power × performance × cost* product capability compared to the its earlier predecessors CMOS 40nm and 65nm, its availability in our testing environment and the availability of enough public information regarding its pricing. The ASIC chip in Figure 3.4 is composed of slave cores, which can either be birthday or neutral-bit slaves. Our digital design flow is shown in below Figure 3.5. Each slave has been synthesized with a top-down strategy using cadence RTL-compiler v14.8, while placement and routing were done using Cadence Innovus. A Power-Aware Synthesis and Placement-And-Routing are used. Power simulations are performed with the pre- and post-placement and routing back-annotated netlist using Cadence Voltus. The slave is then imported as hard macro in Cadence Virtuoso and instantiated from the top-level RTL. The slave and the interface are then placed and routed in Virtuoso GXL.

Figure 3.5.: Our Bottom-Up ASIC digital design flow

The power rail and clock tree are routed with large tracks from the closest power supply and clock pins so as to reduce local voltage drop effect. The RC parasitic extraction of the BDC core GDS and final layout is done using Cadence QRC.

3. On The Cost of ASIC Hardware Crackers

Mentor CalibreDRV is then used for the sign-off DRC and LVS checks. Our design mixes both Regular-Vt (RVT) and Low-Vt (LVT) cells. LVT cells are used without poly-biasing (PB0) for the critical path. RVT cells with poly-biasing up to PB16 are used for the rest of the circuit in order to minimize the leakage power.

Nominal process variation for both PMOS and NMOS for the pre/post-placement-and-routing power simulations with 0.92V supply voltage at 25 degree Celcius are used as parameter for the high performance version of our design. The circuit is first synthesized to reach the maximal operating frequency. Our high speed version reaches 909MHz with $V_{fb} = 0V$ and 1262MHz with $V_{fb} = +2V$.

LVT cells have then been chosen for the critical path of the BDC core. The rest of the circuit have been synthesized with RVT cells so that to balance the performance and power consumption. Each slave is isolated using triple-well isolation to reduce parasitic substrate noise between the slaves that reduces the overall performances. Power simulations show that our $16mm^2$ die requires 140 power supply pins and a plastic-ceramic package to dissipate the power. The effect of body biasing on power and delay after place and route is simulated using Cadence Genus and Voltus. Parasitic extraction with QRC is done with typical parameters. The performances and power result are provided in Table [3.2](#).

3.4.3. ASIC Fabrication and Running Cost

Estimating the cost of fabricating and running an ASIC cluster can be challenging as many parameters are confidential to the fabs. In order to estimate the costs of the attacks considered, we developed a methodology based on the information available publicly. We considered the FD-SOI 28nm technology from ST-Microelectronics. For small scale academic projects, the price of a small batch of up to 100 die, the fabrication cost in US \$ can be estimated by:

$$p_{100} = \begin{cases} 125400 + (A - 12) * 7700, & \text{if } A > 12mm^2 \\ 20900 + (A - 2) * 9900, & \text{if } 2mm^2 \leq A \leq 12mm^2 \end{cases}$$

where A is the die area in mm^2 and p_{100} is the price of the first 100 die in US Dollars (USD). For small scale projects with more than 100 die, the price for a lot of 100 extra die is between 21,120 and 38,500 USD depending on the die area. For our purposes, we consider a small scale project to be a project with at most 25 wafers [\[72\]](#). For large scale projects, a market study published at the FDSOI Forum in 2018 showed that the die manufacturing cost per $40mm^2$ is 0.9 USD for the 28nm technology [\[73\]](#). Hence, our methodology for estimating the costs consists of the three parts we explained. In reality, a more accurate methodology is probably available for the fabs to fill in the gaps. However, we believe that the overall cost will be in the same range.

On top of the fabrication cost, we need to consider the running cost of the ASIC cluster, which includes the energy consumption and cooling. We have performed post-layout extraction and simulation in order to estimate the power consumption of the different chips. In order to simplify the cost analysis, we use a figure of 18 cents/KWh, which is higher than the electricity consumption price in most countries [\[74\]](#). Hence, we only consider the energy consumption of the chips and not the cooling cost or other factors that will be added after fabrication.

3.4.4. Results

Two different architectures of SHA-1 crackers are compared here. The first architecture is based on 2 separate ASIC slaves that handle the two parts of the attack, *i.e.*, the birthday search (BD) and the neutral bits part (NB). The two phases are performed sequentially. Figure 3.6 depicts the overall cost required to build the machine and find the first chosen-prefix collision depending on the time ratio between the two phases. For ASIC, the overall minimum cost is not perfectly at 50% ratio. Hence, we consider a two-stage pipeline architecture at the cost of slightly more hardware to balance the birthday and neutral-bit parts.

Figure 3.6.: Impact of the BD/NB time ratio on the cost

Our birthday (BD) core uses 16927.1 gate equivalents (GEs) per SHA-1 rounds., while our neutral-bit core (NB) uses 170442.7 GEs. Our best implementation is a 4-round SHA-1 unrolled compression function that can be clocked at 900 MHz at $V_{core}=0.92V$ and $V_{fbb}=0V$. Using body biasing and LVT transistors for the critical path, we can further decrease the threshold voltage and increase the running frequency. With $V_{fbb}=+2.0V$ we can increase the running frequency of our fastest core by 40%, reaching 1262 MHz with a 2% increase in dissipated power. The chip can be further over-clocked by increasing V_{core} but at the cost of a quadratic increase in the dissipated power, so a more costly cooling system. The results of our implementations are shown in Table 3.2.

In our study, the overall cost is calculated without the cooling and infrastructure. Note that as shown in Figure 3.7, the total cost required to build an ASIC-based cracker greatly depends on the die size. This is due to the fact that the initial cost is predominant when the die size is large. The overall hardware cost tends to the same for any die size when the attack is fast.

3.4.5. Attack Rates and Execution Time

As shown in Table 3.3, a single NB slave of $16mm^2$ contains up to 24 NB cores and can generate up to 976 solutions up to step 40 of SHA-1 per second. Each

3. On The Cost of ASIC Hardware Crackers

Figure 3.7.: Impact of the die size and latency on the HW cost (4 to 100 mm^2 28nm FD-SOI). The top left line in blue represents 4 mm^2 and the bottom left is 100 mm^2 .

Table 3.2.: ASIC implementation performances for 2 corners cases : high performance at 900 MHz and high performance with FBB at 1262 MHz.

Version	900 MHz		1262 MHz	
	$V_{fbb}=0V$		$V_{fbb}=+2V$	
	BD	NB	BD	NB
Power (in mW)	71.1	289	72.6	294
CP delay (in ps)	1110	1110	792	792
Area (in mm^2)	0.0650	0.6545	0.0650	0.6545

solution A_{40} requires 31 Million cycles, on average. A single BD slave of $16mm^2$ contains up to 245 BD cores and provides a hash rate of 20.6 GH/s for the fastest version of our design. As a comparison, as shown in Table 3.4 and taken from [33], a single GTX 1060 GPU provides a hash rate of 4.0GH/s and can generate 2000 A_{40} solutions per second. If we take the birthday part of the attack as a reference, the neutral bit part is ten time less efficient in hardware than on GPU.

The second architecture is based on GPU. For GPU, it is cost-wise more interesting to take advantage of its reconfigurability to minimize the cost. Hence, we consider in our cost analysis that the chosen-prefix collision is performed serially by reusing the same GPU for the two attack phases. In Table 3.5, the cost of the three attack scenarios is provided. We give in this table the cost to build the ASIC- and GPU-based clusters for 3 different speeds, *i.e.*, one attack per month, one attack per day and one attack per minute. The latency corresponds to the delay to get the first collision. For instance, a two-stage ASIC-based machine able to generate one SHA-1 collision every months, will generate the first collision in

Table 3.3.: Our best $16mm^2$ ASIC implementation performances for 2 corners

Parameter	900 MHz	1262 MHz
SHA-1/core/sec	$2^{25.8}$	$2^{26.3}$
SHA-1/core/month	$2^{47.1}$	$2^{47.6}$
SHA-1/chip/month	$2^{55.1}$	$2^{55.6}$
A_{40} Solutions/core/sec	$2^{4.9}$	$2^{5.3}$
A_{40} Solutions/core/month	$2^{26.1}$	$2^{26.7}$
A_{40} Solutions/chip/month	$2^{30.8}$	$2^{31.2}$

Table 3.4.: SHA-1 hash rate from hashcat for various GPU models, as well as measured rate of solutions at step 33 (A_{33} -solutions). Data taken from [33].

GPU	Arch.	Hash Rate	A_{33} rate	A_{40} rate	Price	Power	Rental
GTX 750 Ti	Maxwell	0.9GH/s	62k/s	250/s	\$144	60W	-
GTX 1060	Pascal	4.0GH/s	470k/s	2k/s	\$300	120W	\$35/month
GTX 1080 Ti	Pascal	12.8GH/s	1500k/s	6.2k/s	\$1300	250W	-

two months. A GPU-based machine generates the first collision in one month for the same attack rate. Our ASIC-based two stage pipelined architecture has twice the latency of a sequential GPU-based machine for the same attack rate. Our benchmark (Figures 3.16 and 3.17) provides a comparison between our ASIC cluster and two of the most widely spread GPU based machines, *i.e.*, the GTX 1080TI (CMOS 14nm) and the GTX 1060 (CMOS 16nm) for different attack rates. The numbers for the GTX 750 TI (CMOS 28nm technology) are also added to the benchmark as it provides an idea of the performance obtained with a GPU based on a similar technology node as our ASIC.

Table 3.5.: Comparison of attack costs with various parameters. Costs are given in USD (k stands for thousand, M for Million, B for Billion, T for Trillion, Q for Quadrillion). Amortized cost is the cost per attack assuming that the hardware is used continuously during three years. Note that it is possible to get slightly more energy efficient platforms and implementations at the cost of more expensive hardware. We list the cheapest platform after one attack, energy included.

Platform	ASIC			GPU rent			GPU buy		
Attack	64	CPC	80	64	CPC	80	64	CPC	80
Energy Cost	\$776	\$1.6k	\$50.9M	-	-	-	\$18k	\$12k	\$1.2B
<i>Cluster for 1 attack per month</i>									
Latency (month)	1	2	1	1	1	1	1	1	1
Hardware Cost	\$257k	\$1.1M	\$11M	-	-	-	\$715k	\$490k	\$47B
First Attack Cost	\$257k	\$1.1M	\$61.9M	\$61k	\$43k	\$4B	\$733k	\$502k	\$48B
Amortized Cost	\$7.9k	\$32.1k	\$51.2M	\$61k	\$43k	\$4B	\$38k	\$26k	\$2.5B
<i>Cluster for 1 attack per day</i>									
Latency (day)	1	2	1	1	1	1	1	1	1
Hardware Cost	\$1.4M	\$3.7M	\$218M	-	-	-	\$22M	\$15M	\$1.4T
First Attack Cost	\$1.4M	\$3.7M	\$269M	\$61k	\$43k	\$4B	\$22M	\$15M	\$1.4T
Amortized Cost	\$2k	\$5k	\$51.1M	\$61k	\$43k	\$4B	\$38k	\$26k	\$2.5B
<i>Cluster for 1 attack per minute</i>									
Latency (minute)	1	2	1	1	1	1	1	1	1
Hardware Cost	\$8.5M	\$48M	\$263B	-	-	-	\$31B	\$21B	\$2Q
First Attack Cost	\$8.5M	\$48M	\$263B	\$61k	\$43k	\$4B	\$31B	\$21B	\$2Q
Amortized Cost	\$781	\$1.6k	\$51M	\$61k	\$43k	\$4B	\$38k	\$26k	\$2.5B

Note on the use of FPGAs Our ASIC design have been tested on FPGA platform. FPGA can be considered as a good alternative to ASIC thanks to its reconfigurability property. However, one of the largest FPGAs from Xilinx, namely the Virtex 7 xc7vx330t-3ffg1157 can fit only 20 instances of the Birthday core running at 135MHz in one chip. The same FPGA can fit only 16 instances of the Neutral Bit core running at 133MHz. In order to do the 2^{64} generic birthday search, we need $2^{36.6}$ FPGA-seconds, *i.e.*, in order to do it in one month we need $2^{15.3}$ FPGAs. As a single FPGA costs around 8000 USD, this attack would cost around 319 Million USD. This is more than one thousand times the cost of the same attack on ASIC and 440 times the cost on GPU, making it irrelevant for the purpose of analyzing SHA-1. Even if FPGAs can be rented, a similar factor is expected compared to renting GPUs.

3.5. Cost Analysis and Comparisons

As explained throughout this chapter, we have performed several experiments to identify the different implementation trade-offs for the attack scenarios we consider. In this section, we analyze the cost estimates of implementing these attacks in ASIC vs. consumer GPU. We consider three attack scenarios that fall into two categories: generic birthday attacks and differential cryptanalysis of SHA-1. Before discussing the analysis in more details, here are a few general conclusions that we reached through our experiments, which can be helpful for building future hardware crackers:

1. The cost of implementing memoryless generic attacks, such as the parallel collision search of [34], in hardware can range from 20% to 50% of the overall ASIC implementation, while the rest is dedicated to the attacked primitive, *e.g.* the SHA-1 hash function.
2. For iterative cryptographic algorithms, such as hash functions and block ciphers, a way to reduce the attack cost is to use unrolling. This approach is similar to using memoryless algorithms. Instead of computing one step of the function every clock cycle, we compute several steps in the same cycle. This amortizes the costs of the attack logic among several steps. For example, implementing the birthday attack using a single-step iterative SHA-1 core leads to a circuits where only 20% of the area is used by the SHA-1 logic and 80% of the area is due to the attack logic, registers and comparisons. On the other hand, using a core that computes 4 steps every clock cycles leads to a circuit with a 50%/50% ratio. While this technique may increase the critical path of the circuit and reduce the frequency, it also reduces the overall number of cycles, so the overall time to compute a single SHA-1 per core is almost constant.
3. For cryptanalytic attacks, the cost is dominated by the attack logic, which may include a huge number of comparisons, modifications and registers. These extra operations are usually different from one step to another, so they consume a huge area. Besides, the state machine of these attacks can be very costly. In such scenarios, the advantage of using ASICs becomes diminished compared to consumer GPUs, except for very high budgets, especially as the GPUs are reusable and can be rented.

3. On The Cost of ASIC Hardware Crackers

3.5.1. 2^{64} Birthday Attack

The first attack scenario we consider is attacking a hash function with 2^{64} birthday collision complexity. The hash function used is the SHA-1 compression function reduced to only 128 output bits, as explained in Section 2.5. A single ASIC core is described in Section 3.2. The time to finish such an attack depends on the number of chips fabricated and the size of each chip. A single ASIC core running at 1262 MHz contributes $2^{26.33}$ SHA-1 computations per second. The attack costs $2^{37.67}$ core-seconds. To reach this complexity, Figure 3.7 shows the price required vs. the estimated time needed to finish the attack, including the fabrication cost of chips of different sizes and the energy consumption.

To put these numbers into perspective, the NVIDIA GeForce GTX 1080 TI GPU (14nm technology) can do about $2^{33.6}$ SHA-1 computations per second, so implementing the attack on GPU would require $2^{30.4}$ GPU-seconds. In order to implement this attack in one month, we need to buy around 550 GPUs costing around 715k USD and around 18k USD in energy. As shown in Figure 3.8, a GTX 1060-based machine is a bit less expensive, costing 525k USD but consuming around 28k in energy for the same job (using 1750 GPUs).

Besides, as shown in Figure 3.8, for any attack rate, it is cheaper to buy an ASIC cluster than a GPU-based cluster. The difference reaches 1 order of magnitude from a rate of 1 attack per week. Furthermore, the ASIC-based cluster consumes 1 to 2 order of magnitude less energy than any GPU-based solution. As shown in Table 3.5, the minimum cost in energy per attack on ASIC is as low as 776 USD. An ASIC-based cracker able to generate one collision per month would cost 257k USD. For an attack rate of 1 attack per minute, it would cost 8.5 million USD.

An alternative option is to rent the GPUs. This would cost around \$61k per attack, assuming a rental price of \$209/month for a machine with 6 GTX 1060 GPUs. This makes the GPU rental very competitive for a single attack, around 4 times cheaper than an ASIC cluster. However, the ASIC cluster quickly become much more cost effective when the attack is repeated (see Figure 3.10).

3.5.2. 2^{80} Birthday Attack

In this section, we look at the cost of implementing a generic birthday collision search for the full SHA-1 output, which requires around 2^{80} SHA-1 computations. The algorithm is the same as the previous attack, except that we use the full output of the SHA-1 compression function. Since a single ASIC core performs $2^{26.33}$ SHA-1 computations per second, the birthday collision search costs $2^{53.67}$ core-seconds, or around 454 million years on a single core. Fortunately, for a powerful attacker with enough money, the cost for producing ASICs grows slowly for large number of chips. The fabrication cost of a hardware cluster to perform the attack in one month costs only 11 million USD, as opposed to around 34 billion USD for GTX 1060. Hence in this case, for any attack rate as shown in Graphs 3.14 and 3.15 the only realistic option is to build an ASIC cluster.

Running the attack costs around 50.9 million USD in energy, which matches the order of magnitude estimated from the bitcoin network: the network currently computes about $2^{70.2}$ SHA-2 every ten minutes, for a reward of 12.5 bitcoin, or roughly \$85k at the time of writing. This would price a 2^{80} computation at 75 million USD.

Figure 3.8.: 2^{64} BD machine price for different attack rates: ASIC vs GPU

Figure 3.9.: Energy cost per 2^{64} BD attack: ASIC vs GPU

3.5.3. Chosen Prefix Differential Collision Attack

The chosen-prefix collision attack proposed by Leurent and Peyrin [33] consists of two main parts: a birthday search attack, and a differential collision attack. The authors provide different trade-offs between the complexity of the two parts. In their paper, the number of solutions required for the neutral bits up to step 33 is provided. This number of solutions corresponds to the number of solutions required to get a valid solution with high probability. Step 33 is chosen because

3. On The Cost of ASIC Hardware Crackers

Figure 3.10.: Total cost (HW+E) for $100 \cdot 2^{64}$ BD attack at a given attack rate: ASIC vs GPU

Figure 3.11.: Total cost (HW+E) for $100k \cdot 2^{64}$ BD attack at a given attack rate: ASIC vs GPU

there is a zero difference at this state, so there is a single path at this step, and solutions are generated fast enough to measure the rate easily. This configuration requires to generate about $2^{62.05}$ SHA-1 computations for the birthday part and $2^{49.78}$ solutions up to step 33. In this chapter, it is cost-wise more interesting for ASIC to generate solutions for the neutral bits up to step A_{40} . There is a factor $2^{7.91}$ difference in the number of solutions to generate between step A_{33} and step

3.5. Cost Analysis and Comparisons

Figure 3.12.: 2^{80} BD machine price for different attack rates: ASIC vs GPU

Figure 3.13.: Energy cost per 2^{80} BD attack: ASIC vs GPU

A_{40} . Hence a chosen-prefix collision requires to generate $2^{41.87}$ solutions. Table 3.4 provides the hash rates and solution rates numbers used in our estimate for the cost on GPU. This gives 38 GPU-years for the birthday, and 65 years for the neutral bits. The estimated cost per attack using GTX 1060 GPU, assuming 209 USD per month for 6 GPU is about 43k USD. The cost of running the attack in GPU is dominated by the energy consumption. ASIC is much more energy efficient, as shown in Figure 3.17. It can be up to 2 order of magnitude less than using common consumer GPU. As shown in Figure 3.16, ASIC-based SHA-1 cracker that

3. On The Cost of ASIC Hardware Crackers

Figure 3.14.: Total cost (HW+E) for 100 2^{80} BD attack at a given attack rate: ASIC vs GPU

Figure 3.15.: Total cost (HW+E) for 100k 2^{80} attack at a given attack rate: ASIC vs GPU

generate one collision per month, costs about 1.1 million USD, about the same as the cheapest GPU-based cracker from our benchmark. However, a single attack on GPU costs about 19000 USD in energy. Hence from 100 attacks as shown in Figure 3.18 and 3.19 as well as for attack rates greater than 1 attack per week, an ASIC-based SHA-1 cracker is the only realistic option.

3.5. Cost Analysis and Comparisons

Figure 3.16.: CPC machine price for different attack rates: ASIC vs GPU

Figure 3.17.: Energy cost per CPC attack: ASIC vs GPU

3.5.4. Limitations

While we did our best to estimate the price of the attacks as accurately as possible, our figures should only be considered as orders of magnitude because the pricing of hardware and energy can vary significantly. ASIC pricing is not completely public, and energy prices depend on the country. Moreover, our estimate only include hardware cost and energy, neglecting other operating costs such as cooling and servers to control the cluster (however, the energy price we use is somewhat

3. On The Cost of ASIC Hardware Crackers

Figure 3.18.: Total cost (HW+E) for 100 CPC attack at a given attack rate: ASIC vs GPU

Figure 3.19.: Total cost (HW+E) for 100k CPC attack at a given attack rate: ASIC vs GPU

high, so it can be considered as including some operating costs).

Another caveat is that we only consider the computation part of the attacks. In reality, there is some need for communication between the nodes, and some steps of the attacks must be done sequentially. Concretely, the generic birthday attacks must sort the data after computing all the chains, and the CPC attack must compute several near-collision blocks sequentially. This will likely add some

latency to the computation, and running the attack in one minute will be a huge challenge, even when the required computational power is available.

3.6. Conclusion

Our study provides a precise comparison between ASIC-based and GPU-based solutions for cryptanalysis, with a case study on generic birthday search and a case study on the recent chosen-prefix collision on SHA-1. For the former, we show that generic birthday attacks can be performed very easily with ASICs against a 128-bit hash function, and that even a 160-bit hash function would not stand against a huge, yet potentially affordable, ASIC cluster. For the latter, we created two independent ASICs that handle the two parts separately. Our comparisons with GPU-based solutions show a clear advantage of ASIC-based solutions. In particular, we remark that the chosen-prefix collisions for SHA-1 can be generated in under a minute, with an ASIC cluster that costs a few dozen Millions dollars. Such ability would allow an attacker to apply the SLOTH attack [64] on TLS or SSH connections using SHA-1.

In the introduction to this chapter, we posed three research questions; the first question is related to the cost of attacks on SHA-1. Our study showed that ASIC is clearly the best choice for very high complexities attacks, or for attacks that need to be performed in a short amount of time. However, for proof-of- concept or cryptographic research in general, where complexities of 2^{64} or less can be computed in a month or so, renting a set of GPUs seems to be the best solution. If the attack needs to be repeated multiple times, or if the speed of the attack is critical, then the initial hardware cost might be amortized and the energy cost per attack might become important. We note that the energy cost will be very high on GPU compared to a dedicated ASIC solution. For a chosen-prefix collision on SHA-1, the energy cost per attack for our speed-optimized ASIC is 1.6k USD. The best GPU based solution from our benchmarks consumes about 12k USD per attack. Hence, the cost of the ASIC-based solution is amortized. Furthermore, when the CPC attack rate becomes higher than 100 attacks per month, the ASIC solution is cheaper than any GPU-based solution in our benchmarks. In this case, the cost of the GPU rent is prohibitive and the ASIC is the only realistic threat.

In the second question, we target the comparison between generic attacks and cryptanalytic attacks for similar theoretical level of numeric complexity. In our study, we show that for a similar level of $\sim 2^{64}$ computations, it is $\sim 75 \sim 82\%$ cheaper to implement a generic birthday search, compared to the differential CPC attack on SHA-1. This means that for these two attacks, the generic attack has an advantage of $5\times$. We need to study more cases, such as the biclique attacks on block ciphers compared to the generic brute force attacks.

Last but not least, the third question is whether the 80-bit security level is still adequate for practical use in less demanding applications. Our study is a warning, showing that not only SHA-1 is indeed practically fully broken, but also that search-based and memory-less generic attacks with complexity $\leq 2^{80}$ are within practical reach.

4. Hardware Performance of the Θ CB3 Algorithm

In this chapter, we study the hardware implementation of SPNs and the Θ CB3 TBC-based mode. The contents of this chapter have been published in the International Conference on Cryptology in India (Indocrypt) 2017 [75].

In order to design a good lightweight TBC-based AEAD scheme, first we need to understand the drawbacks of the state-of-the-art schemes. At the time of writing, the state-of-the-art TBC-based AEAD mode is Θ CB3, proposed by Ted Krovetz and Phillip Rogaway in 2011 [10]. It is a generalization of the OCB BC-based AEAD mode by the same authors. It was proposed as way of analyzing OCB3 using a TBC design based on an underlying BC, *e.g.* AES. Due to its attractive features, some designers suggested using the Θ CB3 mode as black-box design, using an ad-hoc TBC, such as in [76]. In 2014, Jean, Nikolić and Peyrin published Deoxys, a family of AEAD algorithms submitted to the CAESAR competition [21]. Deoxys consisted of two sub-families, Deoxys-I and Deoxys-II. The former is an instantiation of the Θ CB3 mode with a new underlying black-box TBC: Deoxys-BC. It can also be seen as an instantiation of the Tweakable Authenticated Encryption (TAE) mode introduced in [9]. Beside the fact that it uses an ad-hoc TBC, Deoxys-I is attractive in two other regards:

1. It is fully parallelisable.
2. It is an online rate-1 mode. It needs only 1 block cipher call to process each block of the message. An online mode can generate the ciphertext on-the-fly without preprocessing all the plaintext.

Due to these features, we study the hardware implementation of Deoxys-I, and parallelisability in hardware in general. In the original proposal [10], the associated data is first processed using the PMAC [77] structure shown in Figure 2.18. Second, the message is encrypted using the structure in Figure 2.17, computing the message checksum in parallel. Finally, the message checksum is encrypted and XOR-ed to the associated data tag to produce the final tag. In both parts of the algorithm, the tweak input to the TBC is never repeated. These two structures, with minor changes, appear also in other encryption modes, such as OTR [78], SCT [11] and CTR [79]. Therefore, the ideas and techniques presented in this chapter can also be beneficial for these other modes. Before describing the architecture, we present observations that inspired the it:

1. The first and second parts of execution do not depend on each other. Consequently, following the Deoxys-I implementation from Poschmann and Stöttinger on the ATHENA website [80], the order can be reversed. This enables one to use the same storage for both the checksum and tag computation.

4. Hardware Performance of the Θ CB3 Algorithm

2. In Figure 2.17 the computations are completely independent, while in Figure 2.18, there is an output dependency between different blocks. Since there is no input dependency, both structures are fully parallelisable. Additionally, a small temporal shift saves the additional storage needed to store the intermediate outputs of the TBC calls. For example, the first block starts at time $t = 0$ and the second block starts at $t = \Delta t$. At $t = T$ the first block is finished and stored in the tag storage. Finally, at time $t = T + \Delta t$ the second block is finished and XOR-ed with the tag, in-place.

4.1. Related Work

The Cryptographic Engineering Research Group (CERG) at George Mason University (GMU), USA, runs and maintains the online platform ATHENa [81] aimed at fair, comprehensive, and automated evaluation of hardware cryptographic cores. One of their projects is the comparison of FPGA implementations of the CAESAR competition candidates. They have also provided high-speed round-based implementations of round 2 candidates of the competition. Among these candidates, several use OCB-like modes: OCB [8], AES-OTR [78], and Deoxys-I v1.41 [21] (which uses the AES-like TBC Deoxys-BC). Deoxys-BC uses almost the same datapath as AES but defines a new tweakkey-schedule that requires a smaller number of gates to evaluate when compared to AES (but with an additional 128-bit tweak input). It also requires a higher number of rounds compared to AES.

The implementations provided by the CERG team are round-based implementations that compute one cipher round per cycle. These implementations are compliant with the CAESAR Hardware API [82], developed for fair comparison among CAESAR candidates. On the other hand, a round-based implementation of Deoxys-I (encryption only) was provided by Poschmann and Stöttinger, that is not compliant with the required API. One of the requirements of the CAESAR Hardware API is to load the encryption/decryption key into the hardware core at most once per message. Since the implementation from Poschmann and Stöttinger [83] does not follow the API, it permits loading the key again with every message block, allowing the designers to get rid of the master key storage, saving 128 flip-flops. They also save 128 extra flip-flops by noticing that during the tag computation, the encryption of checksum can be computed before the associated data. This enables using the same storage for the message checksum and the intermediate tag value, saving 128 more flip-flops. We follow the latter approach in our implementation due to its obvious area advantage.

4.2. Proposed Architecture

The proposed high-level architecture is shown in Figure 4.1. For simplicity, only the encryption datapath is drawn. However, a similar datapath for decryption can also be included.

The architecture consists of a single-round of the underlying BC, which is divided into N stages, each stage takes one cycle to be processed. If the block cipher requires r rounds, the architecture loads and processes N blocks, every $r \cdot N$ cycles, which leads an average latency of $r \cdot N$ cycles, each cycle has about $1/N \times$ the critical path of a single-round architecture. The overall latency is expected to be

Figure 4.1.: Multi-stream Θ CB3 hardware architecture

equivalent to a simple single-round implementation. The selection of N depends on several considerations:

1. This architecture is intended for high-speed over long messages. It is noticeable that any number of blocks smaller than N requires the same amount of time to be encrypted. Consequently, a very large N leads to a huge overhead for short messages or for messages whose block length is not divisible by N .
2. In order to minimize the key scheduling overhead, especially for FPGAs, it is performed in only one pipeline stage and then shifted N cycles. This is based on the Shift Register Look-up table (SRL) feature of the FPGA LUTs, which allows the utilization of very compact serial shift registers using logic LUTs. For most FPGAs, a single LUT can implement either a 16-bit or 32-bit SRL, which we consider as the upper bound on the value of N .
3. The pipeline registers can add a huge overhead over the simple round implementation. Therefore, in Section [4.3.2](#) we describe a technique to select the optimal locations of the pipeline registers in the FPGA implementation.

From these three considerations, we performed experiments to find the optimal value of N . We found that the optimal value for N is between 2 and 4, neglecting the control overhead, *e.g.* control signals, finite-state machines, etc. If N is greater than 4, the pipeline stages become very small, and the gain from pipelining is offset by the cost of implementing the pipeline registers. This leads to an expected speed-up between $2\times$ and $4\times$. Additionally, for applications that require ultra high-speed over very long messages, *e.g.* disk encryption, high-speed multimedia

4. Hardware Performance of the $\Theta CB3$ Algorithm

interfaces, etc., and do not care about the area, the same architecture can be unrolled into a fully pipelined implementation. This can lead to a huge increase of the throughput. Specifically, the single-round multi-stream architecture requires about $r \cdot N \cdot \lceil \frac{B}{N} \rceil$ cycles to compute B blocks. On the other hand, a fully unrolled architecture has an initial latency of $r \cdot N$ and a new block is generated every cycle, leading to a total number of cycles of $r \cdot N + B - 1$. The speed up over the round-based implementation is given by

$$G = \frac{r \cdot B}{r \cdot N + B - 1}$$

and for very long messages, the unrolled architecture has a speed up of r times. Since the area increases less than r times (only the round part is replicated while the tag and control part have almost the same area), the efficiency is expected to be slightly better. In Section 4.4.1 we show that an AES round can be implemented with a clock frequency greater than 700 MHz on FPGA for $N = 4$, with almost the same number of slices/LUTs as the single-stream case. Therefore, we estimate that this variant can be suitable for applications that require very high-speed authenticated encryption.

4.3. Multi-Stream AES-like Ciphers

AES [26] is a 128-bit block cipher, standardized in 2001 by the NIST. It is based on a Substitution-Permutation Network (SPN). The internal state of the cipher can be viewed as a 4×4 matrix of bytes. It consists of 10 SPN rounds. Each round includes a SubBytes operation for the non-linear part, ShiftRows and MixColumns for the linear permutation and AddRoundKey for the key addition. SubBytes consists of 16 independent instances of an 8-bit Sbox, ShiftRows shifts the bytes in each row, independently, and MixColumns applies a diffusion matrix to each column, independently. All byte operations are done in $GF(2^8)$.

In this section, we quickly review state-of-the-art high-speed AES-128 FPGA implementations (we only discuss full-width round-based and unrolled implementations). A detailed survey on AES datapaths for FPGA is provided in [84]. Full-width FPGA implementations of AES are either unrolled implementations [85], round-based single-stream [86, 87] or round-based multi-stream [88]. Although the scope of this chapter is round-based multi-stream implementations, the optimizations described in this section can be used for any of the aforementioned implementations. In [88], the authors proposed the AES datapath shown in Figure 4.2. Each box in Figure 4.2 represents a pipeline stage, and it can be noticed that the selection of the pipeline stages is based on the functionality of each stage, which leads to two very fast stages in the beginning, then two slow stages afterwards, as the first two stages consists of only a 128-bit multiplexer and a 128-bit XOR gate respectively, while the latter two stages consist of more complex circuits. This limits the maximum possible frequency. In the next sections, we will show why this architecture might not be optimal and describe a new four-stream datapath designed for FPGA to achieve higher performance efficiency.

Figure 4.2.: The AES encryption datapath from [88]

4.3.1. FPGA LUT-based Optimization of Linear Transformations

Generally, circuit optimization consists of two phases: logic synthesis and technology mapping. For certain target technologies, such as FPGA, logically optimized circuits do not provide the optimal mapping to the underlying technology, leaving behind a lot of under-utilized hardware resources. This phenomenon is obvious in the AES Sbox circuits proposed by Boyar [89, 90], which are logical optimizations of the circuit proposed by Canright [91]. These circuits are much smaller than the straight-forward Read-Only Memory-based (ROM) Sbox in terms of gate count and circuit depth. These two features make them the natural choice for low-area ASIC implementations of AES. Interestingly, on the other hand, practical results show that one can achieve a smaller area on FPGA by using the ROM approach [84]. By analyzing this result, it appears that due to the specific details of these circuits [89, 90, 91], it is hard to map them efficiently to look-up tables (LUTs), which are the building block of FPGAs. This leads to a lot of under-utilized/unusable logic gates inside the FPGA.

In Figure 4.3, the number of LUTs required for implementing two 8-bit to 8-bit ROM-based Sboxes (which are both the forward and inverse AES Sboxes) is compared with the implementation of Boyar’s shared encryption/decryption Sbox [89]. These results are plotted against the technology evolution of Xilinx FPGAs as an example. Analysing the chart, it is clear that after the introduction of the Virtex 5 family, logic optimization of the Sbox stopped being beneficial. The reason for that was the introduction of the 6-input LUTs, which enabled implementing an 8-to-1 look-up table using only four 6-input LUTs and three dedicated multiplexers, or five 6-input LUTs. In other words, the ROM-based Sbox has become both faster and smaller than the logic-based Sbox, even when both encryption and decryption are implemented using a shared datapath. While the technology seems to be saturated around the 6-input LUT structure, a hypothetical family has been added to the chart assuming 8-input LUT structure, showing that such a family will make the cost of both logic-based and ROM-based implementations exactly the same (8 LUTs). While these results may seem specific to Xilinx FPGAs, other vendors, e.g. Altera, also use 6-input LUTs as their building blocks and will follow the same trend. Besides, the FPGA industry seems to be saturated around this building block and we believe that the same trend will follow for the upcoming years.

4. Hardware Performance of the $\Theta CB3$ Algorithm

Figure 4.3.: Evolution of the AES Sbox/ISbox area vs. Xilinx FPGA families

We study the application the same techniques to other parts of the AES circuit. The AES `MixColumns` circuit is a matrix multiplication operation over $GF(2^8)$ of the AES state byte matrix by a constant matrix M given by

$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix}$$

which is a circulant MDS matrix. For AES 128-bit architectures, the `MixColumns` operation can be viewed as 16 dot-products of the vector $\begin{bmatrix} 2 & 3 & 1 & 1 \end{bmatrix}$ and a vector composed by a permutation of 4 state words. It can also be viewed as four 32-bit to 32-bit mappings (four matrix-vector products over state vectors). The later view is favorable for ASIC implementations, as it allows reducing the required number of gates by sharing many intermediate results of the computation. Specifically, recently maximov showed that only 92 XOR gates are required for implementing the 32-bit mapping [92]. However, as discussed earlier, since modern FPGAs use big 6/5 input LUTs to implement logic circuits, having a lot of small shared 2/3-input gates is not the most efficient solution. Synthesizing the circuit used in [25] or [93] for Virtex-6 FPGA requires 41 LUTs for low-area and 44 LUTs for high-speed. On the other hand, the dot-product view is given by

$$p = 2 \cdot a \oplus 3 \cdot b \oplus c \oplus d$$

which can be decomposed into

$$\begin{bmatrix} a_6 & a_5 & a_4 & a_3 & a_2 & a_1 & a_0 & 0 \\ 0 & 0 & 0 & a_7 & a_7 & 0 & a_7 & a_7 \\ b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 & 0 \\ 0 & 0 & 0 & b_7 & b_7 & 0 & b_7 & b_7 \\ b_7 & b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 \\ c_7 & c_6 & c_5 & c_4 & c_3 & c_2 & c_1 & c_0 \\ d_7 & d_6 & d_5 & d_4 & d_3 & d_2 & d_1 & d_0 \end{bmatrix}$$

where the elements of each column represent the inputs of one output function. From this perspective, it can be seen that 5 outputs can be implemented using one 5-input LUT, while 3 outputs can be implemented using 7-input LUT, which can in turn be implemented using two 6-input LUTs. That adds up to a total of 11 LUTs per output coefficient, 44 LUTs per output column. This shows that logic optimization does not offer much gain over the straightforward implementation of the transformation. Besides, a deeper look at the view given by the decomposition shows that the three outputs that need 7-input LUTs share two inputs bits, namely a_7 and b_7 . The dot product can be rewritten as can be written as

$$\begin{bmatrix} a_6 & a_5 & a_4 & a_3 & a_2 & a_1 & a_0 & a_7 \\ b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 & b_7 \\ 0 & 0 & 0 & x & x & 0 & x & 0 \\ b_7 & b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 \\ c_7 & c_6 & c_5 & c_4 & c_3 & c_2 & c_1 & c_0 \\ d_7 & d_6 & d_5 & d_4 & d_3 & d_2 & d_1 & d_0 \end{bmatrix},$$

where $x = a_7 \oplus b_7$. This decomposition can be implemented using eight 6-input LUTs and one 2-input LUT, for a total of 9 LUTs per output coefficient, 36 LUTs per output column (which is smaller than the best-reported implementations) or 1.125 LUTs per output bit. It is worth mentioning that this number is near-optimal for any linear transformation over 32 bits, as the optimal number is 1 LUT/bit, which corresponds to the transformation where each output bit depends on n bits, where $2 \leq n \leq 6$ (the case where $n = 1$ corresponds to an identity function and can be neglected, w.l.o.g.). In fact, each 6:1 LUT can be implemented as a 5:2 LUT with shared inputs. Using this feature, our circuit can be indeed implemented using only 8 LUTs, which is the optimal figure. However, this is only a practical result specific to certain FPGAs.

The optimization of the AES inverse MixColumns circuit is less straightforward, as M^{-1} includes more complex coefficients. M^{-1} is given by

$$\begin{bmatrix} E & B & D & 9 \\ 9 & E & B & D \\ D & 9 & E & B \\ B & D & 9 & E \end{bmatrix}$$

A lot of work has been done on how to reuse the same circuit from M to implement

4. Hardware Performance of the Θ CB3 Algorithm

M^{-1} with minimal overhead. This is done by using any of the following relations $M^{-1} = M^3$, $M^{-1} = M \cdot N$ or $M^{-1} = M \oplus K$, where N and K are matrices with low coefficients. In that direction, the circuit given earlier will also be the smallest and the same reasoning can be used to achieve small area for both K and N . However, this approach is most useful for low-area serial implementations with shared encryption/decryption datapath. They do not achieve the best results for high-speed round implementations with dedicated decryption datapath. For example, using $M^{-1} = M^3$ requires 3.375 LUTs/bit and produces a large-depth circuit (low performance), while using $M^{-1} = M \oplus K$ is even larger. The most promising approach is $M^{-1} = M \cdot N$ which requires 288 LUTs/block, corresponding to 2.25 LUTs/bit, which is still far from optimal. On the other hand, the straightforward implementation of M^{-1} leads to output functions that include 19 input bits, which can lead to very low performance. Here, we give a circuit that requires 60 LUTs per output column, corresponding to 1.875 LUTs/bit. First, we convert the dot product to

$$p' = E \cdot a \oplus B \cdot b \oplus D \cdot c \oplus 9 \cdot d = F \cdot (a \oplus b \oplus c \oplus d) \oplus (a \oplus 4 \cdot b \oplus 2 \cdot c \oplus 4 \cdot d \oplus 2 \cdot d)$$

Second, two observations are made

1. $F \cdot (a \oplus b \oplus c \oplus d)$ is constant across any output column.
2. $4 \cdot (a \oplus c)$ is shared by two output coefficients. The same is valid for $4 \cdot (b \oplus d)$.

Using these two observations, a circuit that requires only 60 LUTs per output column can be implemented. The circuit diagram is given in Figure 4.4. Given that MixColumns is one of the main differences between the AES encryption and decryption datapaths, optimizing this primitive is crucial. On the other hand, since 1.875 LUTs/bit is still far from the optimal 1 LUT/bit figure, there may be some room for further optimization.

Figure 4.4.: FPGA-friendly inverse MixColumns circuit

4.3.2. Zero Area Overhead Pipelining

Pipelining has been used by hardware designers/architects as a tool to increase throughput/run-time performance for a long time. However, a fully pipelined block cipher implementation can be costly, due to the large area requirements. A more realistic approach is to use multi-stream implementations. These implementations

start from a sequential implementation that processes one block in C cycles, and divides it into N pipeline stages. This leads to computing x blocks in $N \cdot C$ cycles, where $x \in \{1, 2, \dots, N\}$. x depends on the number of independent block streams the user can leverage. However, this is a double-edged weapon, due to the following reasons:

1. The time required to process one block in a sequential implementation is $\sim C \cdot T$, where T is the critical path delay of the implementation. If the N pipeline stages divide the critical path evenly into segments of $\frac{T}{N}$ delay, the time required to process N blocks becomes $T + t$, where t is a small overhead, leading to $\sim Nx$ speed-up. Unfortunately, the critical path is usually not evenly divided, leading to a sub-optimal speed-up ($< N$).
2. Modern FPGA families consist of a basic building block called LUT6, which is a 6-input single-output look-up table. Additionally, each unit of this building block has an associated Flip-Flop, which the designer/synthesis tool can choose to either use it or not. In Figure 4.5, we show the optimal utilization of a LUT6 unit in a pipelined architecture, where it is used to implement a 6-input circuit followed by storing the output. On the other hand, in Figure 4.6, a poor selection of the location of pipeline stage is in-place, leading to the utilization of 3 look-up tables, instead of 1 in the case of Figure 4.5. In other words, the poor choice of where to place the pipeline registers leads to a significant increase in area.

Figure 4.5.: Optimal pipeline selection

While the impact of moving the flip-flops 1 logic level forward in the previous example is obvious, the designers usually do not have an accurate estimation of the exact LUT utilization before synthesis. Consequently, the designers choose the pipeline stages based on the logical functions, e.g. Sbox, MixColumns, input selection, etc. In our work, we follow a different approach. First, we synthesize a single stream sequential implementation of the required block cipher. Second, we study the output layout to determine the precise distribution of pipeline stages without affecting the structure of the utilized LUTs¹

¹The term zero overhead refers to the number of LUT-FF pairs, as this is the important metric, not the number of LUTs or FFs.

4. Hardware Performance of the Θ CB3 Algorithm

Figure 4.6.: Sub-optimal pipeline selection

4.4. Implementations and Results

Given the aforementioned techniques and optimizations, we give implementations of the AES datapath, Deoxys-I and the LED cipher.

4.4.1. Two-Stream and Four-Stream AES Implementations

Figure 4.7.: Two/Four-Stream AES round datapath implementation

Using the techniques described in Sections 4.3.1 and 4.3.2, we have implemented two multi-stream AES datapaths (two and four streams), shown in Figure 4.7. In addition to the use of the low-area MixColumns circuit and ROM-based Sbox, the locations of the pipelining registers have been selected specifically to ensure a LUT utilization as efficient as possible. In other words, both two-stream and four-stream implementations use the same number of logical LUTs (944 LUTs, without key scheduling), out of which 95% (896 LUTs) are 6-input LUTs. The results are given in Table 4.1. The comparison is restricted to full-width implementations that do

4.4. Implementations and Results

not utilize any BRAMs. For that reason, the implementation in [94] is not included. While it has a very high efficiency (yet smaller than ours, ~ 30 Mbps/slice), this number is biased due to the use of BRAMs to reduce the number of slices. It can be observed that our datapath, with only two streams, outperforms the datapath from [88] (with four streams) in terms of both efficiency and area and reaches the same throughput. This result is achieved due to more than one factor:

1. The critical path in the implementation from [88] consists of three levels of logic inside the Sbox used (one LUT6 followed by MUX7 and MUX8). In our design, the critical path also consists of three levels of logic (Sbox part 2 (LUT6) and the MixColumns circuit proposed in Section 4.3.1 (LUT3 + LUT6)).
2. The MixColumns circuit used is smaller.
3. The first two pipeline stages in Figure 4.2 necessitate the use of 256 LUTs. The two stages altogether can be viewed as a 6-input function, which can be easily merged into a single stage of 128 LUTs (of type LUT6).

Table 4.1.: Implementation results of AES on Virtex-5/6 FPGA (encryption only)

FPGA Family	Implementation	Key schedule	Number of Slices	Max. freq. (MHz)	Throughput (Gbps)	Efficiency (Mbps/slice)
Virtex 5	Ours/2 streams	Offline	347	350	4.5	12.90
	Ours/4 streams	Offline	347	625	8.0	23.00
	[88]/4 streams	Offline	400	350	4.5	11.20
	[85]/ unrolled	Offline	3579	360	46.0	12.88
Virtex 6	Ours/4 streams	Offline	347	752	9.6	27.66
	Ours/4 streams	No	247	752	9.5	<u>38.46</u>
	[85]/ unrolled	Offline	3121	501	64.1	20.55

Table 4.2.: Implementation results of AES on Virtex-5 FPGA (decryption only)

Implementation	Key schedule	Number of Slices	Max. freq. (MHz)	Throughput (Gbps)	Efficiency (Mbps/slice)
Ours/4 streams	No	294	477	6.1	<u>20.7</u>
Ours/4 streams	Offline	445	477	6.1	13.7
[88]/4 streams	Offline	550	350	4.5	7.6

4.4.2. Round-based Two-Block Deoxys-I-128

As mentioned earlier, the Deoxys-I CAESAR candidate uses an underlying TBC called Deoxys-BC. The variant used in Deoxys-I uses a 128-bit block and 256-bit tweakey. The tweakey is divided into 128 bits for the secret key, 64 bits for the key and 64 bits for the block counter and other auxiliary control signals. It is similar to AES, with three major differences:

1. It consists of 14 rounds instead of 10.

4. Hardware Performance of the Θ CB3 Algorithm

2. The final round includes a MixColumns operation, as opposed to AES.
3. It uses a different key schedule, which is smaller than the AES key schedule, but uses an extra public tweak value.

Based on the architecture proposed in Section 4.2 and the AES datapaths proposed in Section 4.4.1, we have implemented two complete datapaths for Deoxys-I. They include two and four pipeline stages, respectively. They also consist of four parts: the encryption datapath, the decryption datapath, the key schedule and the tweak schedule. Using the pipeline selection technique, both implementations consume the same area, except for the decryption data path which we implement only for 4 streams. In Table 4.5, it is shown that the bottleneck of the design, not considering the control overhead, is the decryption data path.

Table 4.3.: Results of the Deoxys-I-128 datapath implementation on Virtex-6 FPGA

Block	Number of LUT-FF pairs	Max. Freq. (2-stream)	Max. Freq. (4-stream)
Enc. datapath	1455	492 MHz	785 MHz
Dec. datapath	1170	-	665 MHz
Key Schedule	442	724 MHz	724 MHz
Tweak Schedule	413	620 MHz	620 MHz

Based on this datapath, a full implementation of Deoxys-I-128 has been realized on Xilinx Virtex-6 FPGA (xc6vlx75tff784-3). The overall utilization is ~ 3800 LUT-FF pairs and maximum operating frequency is 454 MHz. This 27% performance degradation from the datapath estimated clock frequency comes from two reasons:

1. While the datapath is very fast, there is still optimization required to the control unit to cope with such speed.
2. Although we choose a small FPGA from the Virtex 6 family, the design is still small compared to the FPGA size, which leads to it becoming I/O dominated, leading to a lot of wiring delays related to the I/O pins. This will not be applicable if the design is used as a part of a larger on-chip system.

To verify the second problem as part of the reason for performance degradation, we also implemented the design for the small Spartan-6 (xc6slx9ftg256-3) FPGA. The maximum operating frequency is 273 MHz vs. 333 MHz pre-layout (only 18% degradation). The results of the Virtex-6 implementation are summarized in Table 4.4. For fair comparison, we have also downloaded and synthesized the Deoxys-I-128 implementation reported on the ATHENA website [80] by CERG team. We only compare the cipher circuits without the overhead of the hardware API. Our results show an efficiency gain of 75% ($1.75\times$) for Virtex 6 and 74% ($1.74\times$) for Spartan-6. Table 4.4 shows the results for the encryption-only implementation. Our implementation is $5.536\times$ more efficient than the implementation by (Poschmann and Stöttinger).

Table 4.4.: Post-layout results of the Deoxys-l-128 implementation on FPGA

FPGA	Impl.	Number of Slices	Max. Freq. (MHz)	Throughput (Mbps)	Efficiency (Mbps/Slice)
Virtex 6	Ours	861	454	3,874	4.5
	[81]	946	285	2,432	2.57
Spartan 6	Ours	1,010	273	2,329	2.31
	[81]	1,032	161	1,373	1.33

Table 4.5.: Post-layout results of the Deoxys-l-128 encryption only implementation on Virtex 6 FPGA

Impl.	Number of Slices	Max. Freq. (MHz)	Throughput (Mbps)	Efficiency (Mbps/Slice)
Ours	566	416	3,549	6.2
[83]	920	161	1.030	1.12

Extending the results to other technologies (ASIC)

Since it can be argued that the techniques and optimizations in this chapter are limited to a certain technology (Xilinx FPGA), we have synthesized both the proposed implementation and the implementation from [81] for ASIC using Synopsys Design Compiler and the TSMC 65nm technology. The results show the proposed implementation has 54% higher throughput and is 38% more efficient. These results are summarized in Table 4.6.

4.4.3. Three-Stream LED Implementation

LED [19] is a 64-bit block cipher based on an AES-like SPN. Its state is a 4×4 matrix of 4-bit nibbles. In this section we focus on the 64-bit key version LED-64. However, the same results can extend to the other variants of LED, since the only difference is the key scheduling part, which can be easily adjusted for this architecture. The (χ^4) round implementation from [95], Section 3.1, has been replicated for Spartan-3 Xilinx FPGA. Using the guidelines from Section 4.3.2, we

Table 4.6.: Synthesis results of the Deoxys-l-128 implementation using TSMC 65nm technology

Impl.	Area (KGE)	Max. Freq. (MHz)	Throughput (Mbps)	Efficiency (Mbps/KGE)
Ours	59.53	847	7,227	121.40
[81]	53.37	549	4,684	87.76

4. Hardware Performance of the $\Theta CB3$ Algorithm

Figure 4.8.: Three-stream (χ^4) LED-64 round implementation

have been able to add two extra pipeline stages at the outputs of the Sbox and the MixColumns operations, as shown in Figure 4.8. In Table 4.7, it is shown that almost all the available flip-flops have been used, increasing both the throughput and efficiency by $2.57\times$ at no additional area cost.

Table 4.7.: Results of the three-stream LED-64 implementation compared to the single stream counterpart on Spartan 3 FPGA.

Impl.	Number of Slices	Number of FFs	Max. freq. (MHz)	Throughput (Mbps)	Efficiency (Mbps/slice)
[95]/1 stream	204	74	98.7	197.35	0.97
Ours/3 streams	204	202	257	514	<u>2.5</u>

4.5. Conclusion

The experiments in this chapter show that parallelizability, pipelining and time-sharing can be used to improve the efficiency of SPN-based BCs, specially when used in mode that support parallelizability. However, the gain achieved does not match the price paid. For example, given four-way parallelizability we could only improve the efficiency of Deoxys by less than 2. In practice, it may not be useful to target parallelizability for hardware implementations. Besides, the design of Deoxys suffers from large area due to both the extra storage required to handle auxiliary values, and the decryption circuit required. However, this depends on the target application. If high throughput is the ultimate target of the designer, then the techniques proposed in this chapter can be helpful. Of independent interest, we provide discussions on the problem of FPGA optimisation and technology mapping.

Part III.

Tweakable Block Cipher Based-Cryptography

5. Arguments for Tweakable Block Cipher-Based Cryptography

As discussed in Section 2.4, a TBC is a variant of a BC that accepts a public tweak to change its behavior. TBC-based AEAD designs have a lot of nice performance and security features that we will discuss in this chapter. First, we give an early history of TBC designs in Section 5.1. Then, we discuss the impact of the TWEAKEY framework in Section 5.2. Last but not least, we give a summary of recent TBC-based AEAD designs and the advantages and disadvantages of such designs in Section 5.3.

5.1. History

The first formalisation of TBCs was done by Liskov *et al.* [9] in 2002. They introduced the concept of public tweak input to the block cipher in an analogy to the random IV or nonce in modes of operation. By introducing such flexibility at the primitive level, more efficient and simpler modes can be designed. These new modes are both easier to understand and easier to analyze. The paper titled "*Tweakable Block Ciphers*", suggested that:

1. TBCs are easy to design.
2. A BC can be easily converted into a TBC.
3. It is easier to design and prove modes based on TBCs.

The end goal is to provide a primitive where the secret key can be fixed for a long time, while the public tweak can be frequently changed, potentially taking a distinct value for each block. Since the tweak is public, a TBC should be secure against *chosen tweak attacks*, where the adversary can fully control the tweak value. In fact, a stronger desired property is that two ciphertext blocks generated using different tweaks should be independent, even if they are generated using the same key and/or same plaintext.

While [9] is considered the first paper to formalize TBCs, the introduction of TBCs dates back to the Advanced Encryption Standard (AES) selection process in 1998, where Hasty Pudding [96], one of the AES competition candidates, included an extra input called *spice*, which is a secondary public key added to the state of the cipher in each round and is expected to be frequently changing. However, the design did not have security claims against adversaries that can control the spice. Over the years, two main approaches for designing TBCs were developed:

1. Ad-hoc TBCs: These are TBCs that are designed from scratch to include the extra tweak input. These ciphers can themselves be built based on new components, *e.g.* Skinny [18], or rely on the design of older well established BCs, *e.g.* Tweakable-TWINE [97] that reuses the same Feistel Structure as TWINE [98].

5. Arguments for Tweakable Block Cipher-Based Cryptography

2. BC-Based TBCs: These are constructions based on conventional BCs. They resemble a small mode of operation that accepts a secret key, a plaintext/ciphertext block and a public tweak and uses a black-box BC to achieve the same properties of the TBC. However, unlike the security of the ad-hoc TBCs, which is usually claimed to be k bits, where k is the number of key bits, the security of these constructions depend on their inherent structure and security analysis. It can be lower than k , e.g. $k/2$ or $2k/3$ bits. It can also often depend on the block size n instead of the key size k . An example of these constructions is the XEX construction used in the OCB and PMAC modes [41] and it has $n/2$ -bit security.

5.2. The TWEAKEY Framework

As we mentioned in Section 2.4, Jean *et al.* introduced the TWEAKEY framework in [20]. In the process of developing this framework, the authors identified a research problem that made older TBC designs vulnerable to some attacks in specific scenarios. Essentially, while on the surface the tweak is public and seems less critical than the secret key, given that the adversary can carefully control the tweak, design-wise its security requirements can be stronger. The TWEAKEY framework considers the tweak and key to be of the same nature, where a BC is designed with a large key space, and that key space can be arbitrarily divided into the tweak space and the secret key space, as long as the key size maintains the required security level. The low-level details of the TWEAKEY framework are out of scope of this thesis and interested readers should refer to the original paper [20].

While studying the TWEAKEY framework, we observe that it provided a paradigm shift for what real world TBCs can do. We investigate the potential advantages of the TWEAKEY framework. This goes, in some cases, beyond the designers original intentions.

By providing stronger security requirements for the tweak, it increases the scope of TBC-based designs by allowing new constructions to be secure, an issue that is explained in the original TBC paper [9]. To give an example, we recall the Davies-Meyer construction for building collision-resistant compression functions, given in Figure 2.12. The construction uses a black-box BC, where the input is divided between the plaintext input and the key input, compressing an $(n + k)$ -bit input into an n -bit output, where n is the block size and k is the key size. One way to adapt this construction to TBCs is to consider the instantiation in Figure 5.1, where the t bit tweak is also used to consume part of the input, in the same way as the key bits, compressing $(n + k + t)$ bits into n bits.

Figure 5.1.: An adaptation of the Davies-Meyer construction using a TBC

The LRW1 is a TBC construction provided in the original TBC paper [9], shown in Figure 5.2. We can show that using LRW1 in the adapted Davies-Meyer construction can lead to very simple collision attacks. For example, as shown in Figure 5.3, the adversary can perform the following attack:

1. Choose a message consisting of three blocks V, M_l and M_r of sizes n, k and t bits respectively, with output H .
2. Choose a k -bit message block M'_l , where $E_{M_l}(V) \oplus E_{M'_l}(V) = \Delta_1$ and $E_{M_l}^{-1}(V \oplus H) \oplus E_{M'_l}^{-1}(V \oplus H) = \Delta_2$.
3. Choose a t -bit message block $M'_r = M_r \oplus \Delta_1 \oplus \Delta_2$.
4. The two messages (V, M_l, M_r) and (V, M'_l, M'_r) represent a successful collision.

Figure 5.2.: The LRW1 construction

Figure 5.3.: Collision attack on the adapted Davis-Meyer construction using LRW1

The reason this attack is possible is because while the LRW1 construction can be secure in the standard TBC model, where the key is secret, it does not behave as a PRP when both the key and tweak are public and can be chosen by the attacker. On the other hand, it is clear that by using the TWEAKEY framework this type of weakness can be eliminated if the TBC behaves as a PRP for any selection of the tweakey. Of course, this argument assumes the TBC is ideal. In practice, it relies on the cryptanalysis of the TBC in the chosen-tweakey model.

Another advantage of the TWEAKEY framework is the simplicity of the security analysis/proofs. For example, many schemes, *e.g.* OCB, use the TBC as a black-box tool during the pre-analysis, leading to a very simple proof. Then, the designers replace the ad-hoc TBC with a BC-based construction. The security proof and bound have to be adapted, usually leading to one or more of the following scenarios:

5. Arguments for Tweakable Block Cipher-Based Cryptography

1. Drop in the security of the scheme (*e.g.* birthday-bound security in OCB).
2. Full-blown attacks as the one described earlier.
3. Complicated security proofs with more assumptions such as the ideal cipher model assumption.

Using the TBCs based on the TWEAKEY framework, this second layer of analysis is eliminated, as the black-box results can be directly used. On the other hand, by maintaining the original goal of making the tweak cheap and easy to update, the designers of the TWEAKEY framework designed an efficient and cheap tweak scheduling algorithm, which also reduces the cost of generating round keys.

5.3. TBC-Based Authenticated Encryption

AEAD is one of the areas where TBCs can be very powerful tools. Most of the BC-based AEAD schemes are either secure only up to the birthday-bound or require complicated constructions in order to achieve beyond-birthday-bound (BBB) security. The root cause of this issue is that for a given secret key K , all the invocations of the same BC are invocations of the same permutation. Hence, if a collision occurs on the the input/output state of the BC, the same input-output pair of the permutation is used twice. In most cases, this can lead to distinguishing attacks or existential forgery attacks. For example, Ferguson showed in [99] that OCB loses its authentication capability once a collision in the internal state occurs. In case of TBCs, these relations can be broken by changing the public tweak. In particular, changing the tweak value should change the permutation completely. Hence, even for the same key, same input and/or same output, given that the tweak value is different, there is no relation between the different TBC invocations. This indeed potentially leads to a significant gain in terms of security, as not only does the adversary need a collision on the cipher state, but also on the tweak input, and in many scenarios getting such collision is made impossible or at least very hard by design. The designer can select the tweaks in a deterministic way such that they either never collide or collide with very low probability. This phenomenon is illustrated in Figure 5.4

One of the early AEAD schemes to exploit these properties is the OCB mode, where the designers build an AEAD mode based on the XEX TBC construction. As a first stage of analysis, they introduce the Θ CB3 mode, which uses a TBC and has full n -bit security. In fact, the security proof of Θ CB3 is very simple and concise [10], where the confidentiality security proof is simply a direct application of the phenomenon we discussed earlier, such that as long as the adversary is nonce respecting (*i.e.*, never querying the same nonce twice for encryption, in opposition to a nonce-misuse adversary), the tweak value is never repeated for two TBC calls and he/she can only apply the brute force key search in order to recover the secret key.

Once the XEX construction is plugged in, the security drops to the birthday bound and the security proof gets more complicated. During the SHA-3 selection process, the SKEIN team proposed an ad-hoc TBC, Threefish, and suggested that it can be used as an ad-hoc TBC in modes like Θ CB3 [76]. In the CAESAR competition [4], several AEAD schemes used ad-hoc TBCs, an effort that ended

BC-based TBC					Ad-hoc TBC				
	T_1	T_2	T_3	T_4		T_1	T_2	T_3	T_4
K_4	P_4	P_4	P_4	P_4	K_4	P_m	P_n	P_o	P_p
K_3	P_3	P_3	P_3	P_3	K_3	P_i	P_j	P_k	P_k
K_2	P_2	P_2	P_2	P_2	K_2	P_e	P_f	P_g	P_h
K_1	P_1	P_1	P_1	P_1	K_1	P_a	P_b	P_c	P_d

Figure 5.4.: The permutation selection behavior as a function of the key and the tweak. The left diagram is for BC-based TBCs where the key of $E_K(\cdot)$ does not change frequently. The right diagram is for ad-hoc TBCs $\tilde{E}_K^{Tw}(\cdot)$

with Deoxys-II being one of the winners of the competition, for defense-in-depth applications, as testament of the security advantages of TBCs. Afterwards, the Skinny TBC [18] was designed, providing a very cheap hardware solution, as a counterpart for the fast software TBC, Deoxys-BC. One area where more research is required is to design a TBC that is efficient for both software and hardware. Recently, the GIFT cipher, which was originally designed for hardware, was shown to be very fast in software as well [100]. A potential solution would be to use the features of GIFT in combination with the TWEAKEY framework to build a TBC. The TGIF submission [101] to the NIST lightweight cryptography standardization project is an example of this approach, where the cipher designers used the GIFT round function as the building block for TBC. The drawback is that the cipher only supports 128-bit tweakeys. However, this is beyond the scope of our work, as we focus on AEAD modes. Our arguments and designs apply to any TBC with the appropriate parameters. Another TBC that has been recently proposed that has competitive hardware and software performance is Tweakable-TWINE [97]. However, it is limited to only 64-bit blocks. For our work, we use the Skinny TBC as a proof of concept.

5.4. Efficiency Function $e(\lambda)$

A secure cryptographic primitive is an important building block for cryptographic solutions. However, if the primitive is not efficient or not performant, it does not offer a practical solution to real world problems. For any cryptographic algorithm, it is essential to benchmark the cost and performance on different platforms and technologies practically. However, the cipher/mode designer needs to be able to estimate such values prior to designing the full algorithm, so as to compare different possible approaches. Hence, we need theoretical metrics to assess different designs,

5. Arguments for Tweakable Block Cipher-Based Cryptography

and these metrics should follow the same trend as their practical counterparts, even if not perfectly exact. There are four main values that we use in this assessment:

1. Rate R : the rate is the theoretical counterpart of throughput (the amount of input data processed in a unit of time). It is the number of n -bit blocks processed by the mode divided by the number of primitive calls required to process them. The amount of data for a mode of operation that uses an n -bit primitive, *e.g.* n -bit BC or n -bit permutation, is measured in terms of the amount of input data to the mode divided by the output size of the underlying primitive. The time is measured in terms of the number of primitive calls required. A mode using an n -bit primitive needs $1/R$ calls for every n -bits of the input.
2. State size S : any algorithm requires a minimum amount of storage in order to operate on any computing platform, whether it is a general purpose processor or a dedicated hardware circuit. This is directly proportional to the cost of the design, especially in lightweight scenarios where the circuit size or software design is already cheap. This metric is, of course, more relevant to hardware platforms.
3. Non-linear width W : this is the bit-width of the non-linear part of the algorithm, *e.g.* the SPN. This estimates the cost of implementing side-channel countermeasures such as threshold implementations, as they are significantly more expensive for the non-linear parts of the circuit, and in some scenarios implementing them for the linear parts is not even necessary [102]. This metric is related to the width of the non-linear part of the circuit and is different from the non-linear gates per bit metric used in ciphers such as LowMC [103], targeting different applications.
4. Security level λ : The bit security offered by the algorithm against various *black-box* attacks. For generality, we consider that the security level applies to both the adversary's computational power and the amount of data required for the attack to work. There are scenarios where the two requirements do not match, by reducing the amount of data allowed to the adversary while increasing the computational power he/she needs. For comparison purposes, we need to simplify the security parameters by considering only scenarios where both the computational power and the amount of data have matching limitations. Besides, there are applications where having mismatched requirements can be problematic, as we shall discuss in Chapter 6.

A skeptical reader may argue that these metrics are not important in the design process and that what matters is only the practical metrics, such as throughput, area, or energy. Our counterargument is that building high performance and low cost into the design from scratch is not an easy task, as fixing issues that cause large area or low performance once the design is finished are complicated. This is not to say that the implementations and practical benchmarking do not matter, but such assessments are required at an early stage of the design. By not carefully taking performance-security metrics into consideration at an early stage of the design, the scheme could require last minute fixes that decrease the performance or increase the overall cost.

Besides, in this chapter we are interested in the efficiency of different design approaches from a theoretical point of view. There can be some discrepancies between theoretical and practical results that usually arise from the good/bad design of the primitive, yet future primitives can close this gap. At the end, any theoretical estimate must be verified using practical implementations, which should be an integral step of designing a new mode and not a task that comes afterwards.

A designer's goal for lightweight cryptography is to increase R and λ while decreasing S and W . In fact, in many applications, including this thesis, we want a certain security level λ and we want to optimize the other three parameters around it. Consequently, we study the efficiency of permutation-based cryptography, BC-based cryptography and TBC-based cryptography from this point of view.

A public permutation is mainly used in a sponge design, *e.g.* Duplex Sponge [104] and Beetle [105], or in an Even-Mansour (EM) cipher, *e.g.* Elephant [106]. First of all, we can exclude the second scenario from our comparisons by showing that it can be at most as efficient as a BC-based variant of the same mode. The EM cipher is vulnerable to time-memory-data trade-off attacks (TMDT) [107]. Hence, given a security level λ , we need at least $n = 2\lambda$, where n is the bit-size of the permutation input/output. On top of that, the key size must be 2λ , since the key size of the EM cipher is at least equal to the block size. This means that the minimum state size required by the design is at least 4λ . In fact, the state size will usually be even higher, in order to store the nonce and other auxiliary materials such as counters. A BC-based scheme vulnerable to a birthday-bound attack also requires $n = 2\lambda$ and $k = \lambda$. Hence, using a permutation in an EM-based design can be considered as a special case of a BC-based scheme, with a larger state size.

A Duplex Sponge construction requires a permutation of size $n = 2\lambda + r$, where r is the rate of the sponge function in terms of bits. Note that the definition of the rate r is a sponge function refers to the input block size and is different from the overall rate R . Assuming there is no auxiliary material for the mode to use, $S = n = 2\lambda + r$, while $R = \frac{r}{2\lambda + r}$. W is the full width of the permutation, *i.e.*, $W = 2\lambda + r$. For two-pass schemes, such as Encrypt-then-MAC, the rate drops to $R = \frac{r}{4\lambda + 2r}$. In standard Duplex Sponge constructions, the key is used during the first permutation call, and never used again, so it does not need dedicated storage. On the other hand, for schemes that use key-based initialization/finalization, such as Ascon or ISAP, the key needs to be stored as it is used multiple times during one execution and the state size increases to $S = 3\lambda + r$. The state size of Duplex Sponge was believed to be a potential lower bound on the state size of sponge based constructions. Modern sponge-type constructions, such as Beetle, use more sophisticated input handling constructions to optimize the rate and state size, reaching $S = W \approx 2\lambda + \log(\lambda)$, and $R \approx \frac{\lambda + \log(\lambda)}{2\lambda + \log(\lambda)}$.

As discussed earlier, in order to counter the birthday-bound, BC-based schemes require either a block size that is larger than the required security level [41] and/or multiple cipher calls per block [108]. Hence, for a scheme with $R = 1$, the best we can hope for with current design techniques is $S = 3\lambda$. In practice, one of the most efficient BC-based constructions is the COFB mode [46], which satisfies $S = 4\lambda$, $W = 2\lambda$ and $R = 1$, given a BC with a block size of 2λ and key size of λ . In this thesis, we give a construction that relies on the related-key security of the BC and the ideal cipher model called Remus-N. It offers a trade-off between S and λ , where one parameter set can reach $S = 3\lambda$, $W = \lambda$ and $R = 1$. Both COFB and Remus-N rely on TBC-based design techniques using a BC-based TBC.

5. Arguments for Tweakable Block Cipher-Based Cryptography

Particularly, COFB can be viewed as a TBC-based scheme using the XOR-then Encrypt (XE) construction, where a secret mask is added to the BC input, and this mask changes for every call.

Unlike BC-based schemes, a TBC-based scheme can be designed such that it does not suffer from the birthday-bound problem. Hence, the minimum possible state size is $S = n + k + t$, where n and k are at least λ bits, and t affects the number of blocks that can be encrypted, *i.e.*, $t \geq \lambda$, such that $S \geq 3\lambda$. The ΘCB3 mode requires an additional block checksum, raising the state size to $S \geq 4\lambda$. In this thesis, we provide TBC constructions that satisfy $S = 3.5\lambda, 3.2\lambda$ and 3λ , all with $W = \lambda$ and $R = 1$. Independently, Naito *et al.* [109] proposed the PFB construction, achieving similar bounds. Recently, subsequent to the work in this thesis, Naito *et al.* [102] proposed the PFB_Plus construction that requires $S = 3\lambda$, all with $W = \frac{\lambda}{\omega}$ and $R = 1$, where ω is a constant chosen by the designer based on the block size of the underlying TBC, typically $\omega = 2$. This optimizes W while maintaining the other metrics. We believe that Romulus, PFB and PFB_Plus represent a surge in the academic research surrounding TBC-based cryptography and open the door for more efficient schemes, satisfying more goals such as nonce-misuse resistance and leakage-resilience.

In fact, we address one of these goals in this thesis also, by providing three cousins to our three TBC-based constructions, that are secure against nonce-misuse adversaries, by applying a two-pass MAC-then-Encrypt approach. Unlike the vast majority of two-pass designs that have a rate $R \leq 1/2$, our constructions allow a rate of $7/11 \leq R \leq 2/3$, while fixing all the other parameters of their single-pass cousins. Whether it is possible to design a similar construction that has similar properties to PFB_Plus remains an open problem. When it comes to leakage resilience, TBCs have already been used to achieve new results, either as a root source of security in Spook [110], or as an integral design element in TEDT [111].

In practice, one of the ways to measure the efficiency of an algorithm in terms of trade-offs is the throughput/area efficiency. We devise a similar metric, which relates the security, area and performance from a theoretical point of view, namely the efficiency function, given by

The Efficiency Function $e(\lambda)$

$$e(\lambda) = \frac{S + 2W}{R}$$

where S is proportional to the area/storage of the unprotected implementation and W is proportional to the cost of implementing side-channel countermeasures, and R is proportional to the throughput.

The designer's goal is to minimize $e(\lambda)$. In fact, $e(\lambda)$ is *almost* inversely proportional (under certain assumptions) to the throughput/area efficiency function for protected implementations against side-channel attacks, using masking or threshold implementations. First of all, the cost protection of a Boolean function depends on the degree of that function, where a non-linear function needs at least 3 shares for a first order threshold implementation, while a linear function needs only 2 shares. Moreover, if the key bits do not depend on the data input and the internal keys are generated linearly, they can be safely unprotected or protected at a much cheaper cost. For example, the overall state size of the protected threshold

5.4. Efficiency Function $e(\lambda)$

implementation (TI) is lower-bounded by $S + 2dW$, where d is the order of the TI. Denoting the area of a single flip-flop by a_{ff} and the area of the logic part of the circuit by a_l , the overall area of the circuit can be estimated by

$$area = (S + 2dW) * a_{ff} + a_l$$

The throughput can also be estimated by

$$throughput = \frac{nR}{t_c}$$

where t_c is the cost of a single primitive call. Hence,

$$\frac{throughput}{area} = \frac{\frac{nR}{t_c}}{(S + 2dW) * a_{ff} + a_l}$$

We assume that:

1. $\frac{n}{t_c}$ is almost constant: the cost of cryptographically generating n bits depends on the value n and does not vary so much by changing the primitive.
2. a_l is small compared to $(S + 2dW) * a_{ff}$ and can be ignored in theoretical comparisons on the mode level. This assumption has been supported by practical evidence, where the implementations of the Remus TBC [18] and our Romulus-N AEAD mode are good examples.

Under these two assumptions,

$$\frac{throughput}{area} \propto \frac{1}{e(\lambda)}$$

Table 5.1 lists some of the modern AEAD constructions and how they measure up against our efficiency function. We can see that TBC-based schemes, whether ad-hoc or BC-based can reach $\leq 5\lambda$ in the single-pass setting and $\sim 8\lambda$ in the two-pass setting. The most efficient TBC-based scheme according to our metric is PFB_Plus, for which $e(\lambda) = 4\lambda$ when $\omega = 2$. Following PFB_Plus, Romulus-N, PFB and Remus-N achieve 5λ . The most efficient misuse-resistant mode is Romulus-M.

Table 5.1.: Comparison between different constructions in terms of performance metrics for a security level λ

Type	Construction	Rate R	State Size S	Width W	$e(\lambda) = \frac{S+2W}{R}$	Misuse-Resistant
TBC	Romulus-N [112]/PFB [109]	1	3λ	λ	5λ	-
	PFB_Plus [102]	1	3λ	$\frac{\lambda}{\omega}$	$(3\omega + 2) \cdot \frac{\lambda}{\omega}$	-
	Θ CB3 [10]	1	4λ	λ	6λ	-
	Romulus-M1 [112]	$2/3$	3.5λ	λ	8.25λ	✓
	Romulus-M3 [112]	$7/11$	3λ	λ	7.86λ	✓
	SCT [11]	$1/2$	4λ	λ	12λ	✓
	ZAE [113]	$1/2$	7λ	λ	18λ	✓
BC	Remus-N [114]	1	3λ	λ	5λ	-
	COFB [46]	1	4λ	2λ	8λ	-
	Remus-M [114]	$1/2$	3λ	λ	10λ	✓
	SUNDAE [115]	$1/2$	3λ	2λ	14λ	✓
Sponge	Duplex Sponge [104]	$\frac{r}{2\lambda+r}$	$2\lambda + r$	$2\lambda + r$	$\frac{12\lambda^2+12r\lambda+3r^2}{r}$ (24λ at $r = 2\lambda$, 27λ at $r = \lambda$)	-
	Beetle [105]	$\frac{\lambda+\log(\lambda)}{2\lambda+\log(\lambda)}$	$2\lambda + \log(\lambda)$	$2\lambda + \log(\lambda)$	$12\lambda + \frac{3\log^2(\lambda)}{\lambda+\log(\lambda)}$	-
	Ascon [6]	$2/5$	3.5λ	2.5λ	21.25λ	-
	Two pass Duplex Sponge	$\frac{r}{4\lambda+2r}$	$2\lambda + r$	$2\lambda + r$	$\frac{24\lambda^2+24r\lambda+6r^2}{r}$ (48λ at $r = 2\lambda$, 54λ at $r = \lambda$)	✓

5.5. Applications and Discussions on the Efficiency Function

As we have discussed, the efficiency function is not a substitute for practical experiments. Hence, a question we need to ask is how to use it correctly without making false conclusions? For example, is it true that in practice a Duplex Sponge construction is 5 times less efficient than a TBC-based construction? The answer to these questions is admittedly not straightforward. To understand the answers, we need first to consider what $e(\lambda)$ measures, which is the efficiency of a construction when built from ideal primitive, *e.g.* an ideal cipher in the case of BC or TBC-based constructions, or ideal permutation in the case of permutation-based cryptography. In other words, it measures the efficiency of the mode as described in its security proof. In practice, such ideal primitives do not exist. However, there is still a distinction between permutations and ciphers. Usually, BCs or TBCs are built to be as close as possible to ideal ciphers, where a strong requirement is that there are no cryptanalytic attacks faster than brute-force key search. In fact, there is usually a 30 ~ 50% security margin between the best known attack on a reduced version of the cipher and the actual real instance of the cipher, and a cipher with a smaller security margin is considered *risky*. This is true for most lightweight (T)BCs, such as AES, GIFT, Skinny, Tweakable-TWINE and SIMON. On the other hand, in permutation-based cryptography, such margin is so narrow, and in many cases non-existent. Designs focus on achieving security in practice, rather than satisfying theoretical assumptions. A designer would choose a non-ideal permutation given that there is no attack on the mode when used with such permutation. In fact, most of the lightweight permutations used in practice are either non-ideal by design, *e.g.* Subteranean, or have been proven to be non-ideal by finding cryptanalytic distinguishers from an ideal permutation, *e.g.* GIMLI, Ascon, Xoodoo and PHOTON-Beetle. This qualitative difference implies that it is hard to compare the two approaches fairly in practice, without having presuppositions. We do not intend to give an easy answer to which approach is better, but to provide the designers with tools to assess their schemes so they can reach the conclusions that fit into their target applications. It is unfair to conclude that a Duplex Sponge construction is 5 times less efficient than a TBC-based design as the goals, assumptions and techniques are vastly different. On the other hand, a designer can use the efficiency function to optimize their scheme by tweaking different parameters to minimize $e(\lambda)$.

Part IV.

TBC-Based Authenticated Encryption

6. Analysis of the Rekey-and-Chain Schemes

In this Chapter we discuss some of the security aspect of (T)BC-based lightweight AEAD schemes. The results in this chapter were published in the IACR Transactions on Symmetric Cryptology (ToSC) 2019, issue 4 [116], and in a note on the ePrint cryptographic archive [117].

One of the main challenges facing the designers of lightweight AEAD schemes is reducing the state size of the algorithm. The state size translates either into memory limitations for software implementations or hardware area for ASIC/FPGA implementations. Most of the time the state size is the main contributor to how small the implementation of an algorithm can be. For example, the low-area implementation of the SKINNY lightweight tweakable block cipher [18] with 128-bit block and 128-bit key is only $\sim 23\%$ smaller than the corresponding round-based implementation. In order to design AEAD schemes based on (T)BCs that have limited state size while meeting certain security goals, three techniques have recently become popular. The first one is chaining, where the input to a block cipher is a linear combination of the previous output of the block cipher and the current input block. This ensures that the input to a block cipher depends on the previous block cipher calls and not just the current input block. This helps reduce the state by making it a function of all the previous plaintext/ciphertext blocks, eliminating the need for an auxiliary checksum. The second technique is rekeying, where at the beginning of the encryption process, a random value is generated based on the master key, and this value is later used as during the block cipher calls, generally as a tweaked key. Essentially, the rekeying function operates as a compression function that uses the nonce, master key and block information (*e.g.* block index) to generate the TBC key, which is usually smaller than the aggregate size of these values. In such scenario, all the TBC keys are related and it is enough to store the first TBC key and use it to generate the other keys. Thus, the nonce and block information do not need to be stored independently. In addition to helping reduce the state size, rekeying is also useful against physical attacks, such as side channel and fault attacks, by limiting the exposure/usage of the master key. The third technique is similar to rekeying: it is called masking. A random value is initially generated based on the master key and a unique *nonce*, then it is used in a construction on top of the block cipher to mask the state and increase the randomness.

While the latter two techniques are relatively similar, a fundamental difference is that rekeying usually requires stronger assumptions about the underlying block cipher as the security of the mode can only be proven in the ideal cipher model as opposed to the (T)PRP Model. Table 6.1 shows some of the current AEAD designs and which of these three techniques they use. The comparison is limited to (online) rate-1 modes that are secure in the nonce respecting model and use chaining, *i.e.*, modes that use almost a single block cipher call per input block and are secure in the nonce respecting model. Table 6.1 does not serve as an

6. Analysis of the Rekey-and-Chain Schemes

Table 6.1.: Some lightweight AEAD designs and the construction techniques/proof assumptions used in them.

Mode	Chaining	Rekeying	Masking	Assumption
Romulus-N1 [112, 118]	✓	-	-	TPRP
COFB [46, 47]	✓	-	✓	PRP
HyENA [119]	✓	-	✓	PRP
Remus-N1 [114, 118]	✓	✓	-	ICM
Remus-N2 [114, 118]	✓	✓	✓	ICM
COMET-128 [120]	✓	✓	-	ICM
mixFeed [121]	✓	✓	-	ICM

exhaustive list. The modes in Table 6.1 represent around 25% of the (T)BC-based submissions to the NIST lightweight cryptography standardization process and 10% of the overall designs. Moreover, they can be considered as part of the state-of-the-art of this field. Hence, the security analysis and claims made by the designers have to be carefully assessed and scrutinized by the community in order to understand them and have a higher trust of their security. In this chapter, we study two potential issues related to these modes:

1. The weak keys problem in rekeying/masking-based schemes.
2. The nonce-misuse resistance, or rather insecurity, of these schemes.

We give a framework to model these modes and analyze weak keys in chained BC-based modes. In fact, the two problems discussed are related, as they both arise when the block cipher keys are repeated, whether in one message (weak keys) or in a pair of messages (nonce misuse).

Besides, we show that, especially for authenticity, the distinction between the single-key and multi-key settings can be blurry and the existence of weak keys can lead to security degradation if they are not handled properly.

We apply this framework to two round AEAD candidates for the NIST Lightweight Cryptography Process: COMET-128 and mixFeed, with brief comments on two other candidates COFB and HyENA.

Our attacks against COMET-128, COFB and HyENA still fall outside the security requirements in the call for submissions of the NIST and hence does not threaten the security within these limits. However, the results are worrisome in the multi-key setting, where many users adopt the scheme and can be targeted by the same attacker, which is a realistic model in scenarios such as IoT and ubiquitous computing. For example, in this model, we show that the master key of at least one of the users of COMET-128 can be recovered with only $\mathcal{O}(2^{64}/\mu)$ online queries per user and computational complexity of 2^{65} by targeting μ users. This means that if only ~ 0.5 million users adopt this algorithm, the scheme is vulnerable to a key recovery attack on one of the users, even if all the users respect the data limits required by the NIST, as shown in Figure 6.1. To put it differently, the NIST requires designs to be secure as long as the data complexity is $< 2^{50}$ bytes and the time complexity is $< 2^{112}$. For COMET-128, in order to operate within these bounds, the number of users must not exceed ~ 0.5 million users, with key recovery

Figure 6.1.: Bit security degradation vs. the number of users for COMET-128

possible as soon as this limit is reached. The number of expected vulnerable keys also grows with the number of users, with a factor of 2^{-19} . For example, if ~ 4 million users respect the data limit of $2^{50} - 1$ bytes, it is expected that 8 users are vulnerable. While it is expected that security drops when increasing the number of users, we believe the data limits on each individual users should not drop, as a user does not know if he is being targeted by a single-user or multi-user adversary. Besides, this security drop is not in the same proportions for all schemes, as proven by Luykx et al. in [122], and 0.5 million users is a very small limit on the number of users. We propose two potential fixes, one with higher performance/area cost, yet more secure, and one with virtually no additional cost and acceptable security.

On the other hand, we show that in the case of **mixFeed** there is a lack of understanding of the underlying primitive that will require extra assumptions in order to prove the security of the scheme even with bit security as low as 50. We give an existential forgery attack of messages longer than 2^{34} blocks, showing that the success probability violates the security claims made in the initial **mixFeed** submission. Subsequently, Leurent and Pernot [124] have addressed some of the questions raised in our attack and have shown that the success probability of the attack proposed against **mixFeed** is significantly higher than originally expected. In fact, they show that the success probability is $\sim 45\%$, disproving both the assumption provided by the designers to address our attack and their integrity security claims. They managed to implement our attack with data complexity of 220 GB bytes.

Besides, we address the security of these schemes in the nonce-misuse scenario.

6. Analysis of the Rekey-and-Chain Schemes

The designers of mixFeed initially conjectured that the design provides integrity nonce-misuse resistance up to a data complexity of 2^{32} . We provide an attack that disproves this conjecture, requiring a single repetition of the nonce and a negligible amount of data.

6.1. Background and Motivation

6.1.1. Weak Key Analysis of Authenticated Encryption

Weak keys are defined as keys that behave in an unexpected manner compared to how it is intended to and detecting whether a secret key belongs to the set of weak keys is easy [125]. For example, if an algorithm requires that every call to the primitive used in that algorithm has a different key, but for some keys the key is fixed for all primitive calls, that is an unexpected behavior. In many cases (as the cases we target in this chapter), this behavior is detectable with one verification query, since if the weak key occurs, it immediately leads to forgery. Usually, the security bounds are calculated on average over the whole key space, which limits the vulnerability of the weak keys in the single-key setting.

Whether the weak keys are exploitable or not is a different problem. An example of a case where the weak keys were exploitable in the AEAD forgery context is the attack on POET by Abdelraheem et al. [126].

6.1.2. Multi-Key Analysis

In [122], Luykx et al. showed that the effect of analyzing the security of a symmetric key primitive in the multi-key setting can lead to drastic security degradation when the same primitive is used by a huge number of users using different keys. They gave a formula for the adversarial advantage gain against a scheme that has weak keys when used by many users at the same time. If the probability that any key is a weak key is p , the probability that all the μ keys (where μ is the number of users) are not weak is given by

$$(1 - p)^\mu$$

and the probability that at least one of them is weak is given by

$$1 - (1 - p)^\mu$$

This probability increases almost linearly when μ is small and approaches 1 when $\mu > p^{-1}$. For example, if $p = 2^{-32}$, the advantage is 0.98 at $\mu = 2^{34}$, 0.63 at $\mu = 2^{32}$, 0.39 at $\mu = 2^{31}$, 2^{-16} at $\mu = 2^{16}$ and 2^{-31} at $\mu = 2$. In [127], Bellare and Tackmann analyzed GCM with respect to multi-key security degradation, establishing security bounds by a factor μ , showing that the success probability of the adversary can be bounded by $\frac{\mu\sigma^2}{2^n}$, where σ is the overall number of resources available to the adversary. This bound is troublesome since it shows that a multi-key adversary may be able to pay less resources per key compared to a single-key adversary. Specifically, if such bound was tight, the per-key cost would decrease by a factor of $\mu^{-3/2}$ in the multi-key setting. Fortunately, Luykx et al. [122] showed that this bound is not tight and that the multi-key security of GCM does not depend on μ once the underlying cipher is replaced with a uniformly

distributed random permutation. More importantly, they provided a sufficient and provable/falsifiable condition for which a scheme would enjoy no security degradation in the multi-key setting. Hence, it is only natural that we need to study newly proposed schemes against this condition and show which schemes are insecure in the multi-key setting.

NIST candidates for the new lightweight cryptography standard have been required so far to be secure only in the single-key setting [3]. Usually, considering only the single-key setting reduces the vulnerabilities related to weak keys compared to the multi-key setting [122]. However, the multi-key analysis is relevant in two ways:

1. Modes such as COMET-128, mixFeed, COFB, HyENA and Remus use rekeying as a technique even in the single-key setting. Hence, we show that the single-key cryptanalysis depends on the multi-key setting.
2. Even if it is not required for the standardization process, understanding and analyzing new schemes in different models that capture realistic cases is a crucial task, given the drastic impact that these new schemes will have if they are actually adopted by the industry.

6.2. COFB-like Schemes

Several other proposals have tried to address the shortcomings of COFB (Section 2.6.3) by designing closely related, yet different schemes, such as: HyENA [119], COMET [120] and mixFeed [121]. As part of the security proof of COFB, the authors introduced the idealized COFB (iCOFB) mode, and most of the examples we mentioned earlier can be viewed as different instantiations of (iCOFB), with some changes to the linear function ρ . While it may be tempting to use the same representation of iCOFB to analyze these modes, it is not the most adequate representation to capture our analysis. Instead, we propose a new representation, which we call Rekey-and-Chain (RaC).

Notation In this framework, each (T)BC has a different (twea)key. The message string M is parsed as m blocks of n bits each: $(M_1, M_2 \dots M_m) \stackrel{n}{\leftarrow} M$. We use the symbol Z_i to refer to the tweakkey of the TBC call after processing the message block M_i , while we use K to refer to the AEAD master key. IV is the initial value of the plaintext input to the first (T)BC call after initialization and processing of AD. Perm is a permutation defined over the (twea)key space. For simplicity, we consider only the case where A, M, C consist of full blocks, *i.e.*, their size is a multiple of n , so that we can neglect the domain separation control logic, padding and length extension attacks which are irrelevant to our analysis. X_i, Y_i are the input and output states of the underlying (T)BC corresponding to M_i/Z_i , respectively. $\text{KDF}(N, K)$ is the nonce-based key derivation function, outputting two bit strings I and J , which is what distinguishes modes that follow the RaC representation. $\text{Absorb}(A, I, J)$ is a function that takes an AD string A and the outputs of the KDF function: I and J , and outputs Z_0 and IV . Given Z_0 and IV and a message M , $\text{Enc}(N, Z_0, IV)$ generates the ciphertext C and authentication tag T . An RaC algorithm consists of three phases

1. $(I, J) = \text{KDF}(N, K)$

6. Analysis of the Rekey-and-Chain Schemes

Figure 6.2.: The RaC representation of COFB-like modes

2. $(Z_0, IV) = \text{Absorb}(A, I, J)$
3. $(C, T) = \text{Enc}(M, Z_0, IV)$

which are elaborated in Figure 6.2. Since it is straightforward to see how each of the modes discussed in this chapter is an instantiation of RaC, we will not go into the details of these instantiations, unless such information is necessary for our analysis. In order for the attacks to work, RaC must satisfy the following properties:

1. Given A , Absorb is easily invertible. In other words, given Z_0 and IV , the attacker can find $(I, J) = \text{Absorb}^{-1}(Z_0, IV)$.
2. ρ is linear and invertible. Additionally, given M_i and C_i , the attacker can calculate Y_{i-1} and X_i . To put more clearly, ρ can be split into two linear functions, $X_i = f_x(M_i, Y_{i-1})$ and $C_i = f_c(M_i, Y_{i-1})$, and given M_i and C_i , the internal states can be calculated using the two linear functions $X_i = f'_x(M_i, C_i)$ and $Y_{i-1} = f'_y(M_i, C_i)$.

6.2.1. Generic Attacks against RaC

Similar to any AEAD scheme, the schemes based on RaC are vulnerable to a wide range of generic attacks. In order to provide a reference point for the reader to compare our attacks to these generic attacks, we provide a non-exhaustive list of examples in Table 6.2.

6.2.2. Forgery attacks against RaC

If Z_0 is a fixed-point of Perm , RaC is vulnerable to forgery attacks. For example, assume that M consists of two blocks. The adversary can apply the following attack:

Table 6.2.: Some generic attacks against RaC. $H(x)$ is the information theoretic entropy of x . We assume that Z_0 and IV are uniformly random. In practice the birthday attack costs may be different if this assumption is not valid.

Attack	Complexity
Master Key Guessing	$\mathcal{O}(2^{ K })$
Tag Guessing	$\mathcal{O}(2^{ T })$
State Guessing	$\mathcal{O}(2^{H(Z_0)+H(IV)})$
State Matching (Birthday Attacks)	$\mathcal{O}(2^{\frac{H(Z_0)+H(IV)}{2}})$
Encryption-Decryption Collision	$\mathcal{O}(2^{\frac{H(Z_0)+H(IV)}{2}})$
Online-Offline Collision	$\mathcal{O}(2^{\frac{H(Z_0)+H(IV)}{2}})$

1. The adversary asks for an encryption query of M , which consists of m blocks, such that $m \geq 2$.
2. Given the corresponding ciphertext/tag pair (C, T) , he/she can calculate the internal state value X_2 .
3. He/she finds a ciphertext block C_x and a plaintext block M_x such that $(X_2, C_x) = \rho(IV, M_x)$.
4. The adversary builds a forgery ciphertext $C_F = C_x \parallel C_3 \parallel \dots \parallel C_m$ and asks for the decryption query (C_F, T) .

If Z_0 is not a fixed-point of Perm , but a member of a short cycle of period l , i.e., $Z_0 = \text{Perm}^l(Z_0)$ for a small value l , then the attack is modified to

1. The adversary asks for an encryption query of M , which consists of m blocks, such that $m \geq l + 1$.
2. Given the corresponding ciphertext/tag pair (C, T) , he/she can calculate the internal state value X_l .
3. He/she finds a ciphertext block C_x and a plaintext block M_x such that $(X_l, C_x) = \rho(IV, M_x)$.
4. The adversary builds a forgery ciphertext $C_F = C_x \parallel C_{l+1} \parallel \dots \parallel C_m$ and asks for the decryption query (C_F, T) .

Another possible forgery attack is

1. The adversary asks for an encryption query of M , which consists of m blocks, such that $m \geq l + 1$.
2. Given the corresponding ciphertext/tag pair (C, T) , he/she can calculate the internal state values X_1, Y_1, X_l and Y_{l-1} .
3. He/she finds a ciphertext block C_x and a plaintext block M_x such that

6. Analysis of the Rekey-and-Chain Schemes

$$(X_1, C_x) = \rho(Y_{l-1}, M_x).$$

4. The adversary builds a forgery ciphertext $C_F = C_1 \parallel \dots \parallel C_{l-1} \parallel C_x \parallel C_{l+1} \parallel \dots \parallel C_m$ and asks for the decryption query (C_F, T) .

However, since Z_0 is secret, applying the attack is not straightforward. Let p_l be the probability that a key Z_0 picked uniformly at random is a member of a cycle of period l , then the attacks described above have a success probability of p_l and data complexity $2l$ blocks for an adversary who assumes Z_0 is a member of such cycle. In order to have a success probability close to 1 the adversary requires roughly a number of encryption/decryption queries $q = p_l^{-1}$ and data complexity of $q(l+1)$ blocks. A very important observation is that this attack does not depend on K , N , A or IV . Hence, an adversary targeting μ users simultaneously can rearrange his resources, spending only $q_k = \frac{q}{\mu}$ queries per key, expecting to achieve at least 1 successful forgery. The adversarial advantages of these attacks are qp_l in the single-key setting and μqp_l in the multi-key setting.

In order for a scheme to be immune to this attack, the designer needs to make sure that $\frac{l+1}{p_l}$ is smaller than the complexities of the generic attacks in Table 6.2, or at least small enough to fall within the targeted security level for every possible value of l .

6.2.3. Analysis of COFB

COFB and HyENA are very similar, with the main difference being in the ρ function. However, in both cases they satisfy our requirements. Hence, the same analysis can be applied to both modes. In this case, the outputs of the KDF are $(L, R) \xleftarrow{\frac{n}{2}} IV$ and $Z_0 = K \parallel L$, where L and R are $n/2$ -bit random variables. $\text{Perm}(K \parallel L) = K \mid \alpha \cdot L$, where α is a primitive element of $\text{GF}(2^{n/2})$. If $L = 0$, then COFB is vulnerable to the forgery attack in Section 6.2.2. In other words, Perm has 2^k fixed-points and the probability of Z_0 being one of them is $2^{-n/2}$. In the single-key setting, this leads to an attack with complexity roughly $2^{n/2}$. However, the designers of COFB and HyENA do not claim security beyond $2^{n/2 - \log(n)}$ online queries with negligible number of offline queries, which means that this attack does not pose a threat against them. In the multi-key setting, the attack requires $\frac{2^{n/2}}{\mu}$ data complexity per key. In scenarios where the number of keys used is relatively high, *e.g.* 2^{24} , and $n = 128$, these modes offer only 40-bit security against forgery over all keys, *i.e.*, with 2^{40} queries per key, one of the users can be vulnerable to forgery.

6.3. Application to COMET-128

COMET-128 [120] is a block cipher based AEAD algorithm submitted to the NIST lightweight cryptography standardization project. It can be described using RaC. Given a BC with n -bit block size and equal key size, *i.e.*, $k = n$, and AD of a blocks, $Z_0 = \text{Perm}^a(E_K(N))$ is a random n -bit value and $J = K$.

The permutation Perm of COMET-128 is depicted in Figure 6.3, where γ is a block constant used to distinguish different phases of the algorithm: Absorb and Enc, and to distinguish incomplete blocks. For most blocks, and for the blocks included in our attacks $\gamma = 0$. The ρ function in COMET-128 is given in Definition 6.1. During the third phase of RaC, *i.e.*, Enc, $\text{Perm}(Z_i) = \gamma \oplus 2 \cdot L_i \parallel R_i$

Figure 6.3.: The permutation Perm of COMET-128

over $\text{GF}(2^{k/2})$, where $(L_i, R_i) \stackrel{k}{\leftarrow} Z_i$. We show that for every pair $(N \parallel A, K)$ there are $2^{k/2}$ weak keys, with probability $2^{-k/2}$ that the Z_0 is a weak key. The existence of these weak keys and the applicability of the multi-key analysis leads to a set of interesting results:

1. After one online encryption query of length at least 32 bytes and one decryption query of length at least 16 bytes, forgery is successful with adversarial advantage $\sim 2^{-64}$. With 2^{64} online encryption queries of 32 bytes each, and 2^{64} decryption queries of 16 bytes each, forgery is successful with adversarial advantage approaching ~ 1 .
2. If the forgery is successful, the master key can be easily identified with high probability with 2^{65} offline queries.

Definition 6.1

Given Y_{i-1} and M_i , $(C_i, X_i) = \rho(M_i Y_{i-1})$ is defined by

$$C_i = \text{Shuffle}(Y_{i-1}) \oplus M_i$$

and

$$X_i = Y_{i-1} \oplus M_i,$$

where $\text{Shuffle}(X)$ is an invertible linear permutation, and $\text{Shuffle}^{-1}(X)$ is its inverse permutation.

Figure 6.4.: A fixed point in the permutation Perm of COMET-128

6.3.1. Existence of Weak Keys

Figure 6.4 shows a fixed point in the permutation Perm of COMET-128. If $L_0 = 0^{64}$, then $Z_0 = Z_1$. Since this event is defined over 64 bits of Z chosen using the KDF (a permutation over $\text{GF}(2^{128})$), there are 2^{64} weak values of Z_0 and the probability that Z_0 is weak is equal to $\frac{2^{64}}{2^{128}} = 2^{-64}$. Since COMET-128 applies the KDF with a different N for every different message and since the KDF is a permutation, given μ online queries, we get μ messages encrypted with μ different values for Z_0 .

6.3.2. Existential Forgery Attack with Weak Keys

Given we have established how the weak keys behave and their probability, we describe how to forge a ciphertext once a weak key has been sampled by the KDF. Let M be the known message encrypted with a weak key Z_0 , where $|M| \geq 256$. The corresponding ciphertext/tag pair is (C, T) . Let M_1 and M_2 be the first two message blocks after parsing M , with C_1 and C_2 as the corresponding ciphertext blocks. Since the attacker knows M and C , he can retrieve the internal state values IV and X_2 , where IV is the state before the absorption of M_1 and X_2 is the state after the absorption of M_2 . Hence, we have

$$IV = \text{Shuffle}^{-1}(M_1 \oplus C_1)$$

and

$$X_2 = M_2 \oplus \text{Shuffle}^{-1}(M_2 \oplus C_2)$$

The attacker wants to find M_x and C_x , such that

$$IV = \text{Shuffle}^{-1}(M_x \oplus C_x)$$

$$X_2 = M_x \oplus \text{Shuffle}^{-1}(M_x \oplus C_x)$$

which is a simple well-defined system of linear equations defined over 256 Boolean variables and easily solvable. This attack has been verified by modifying the reference implementation of COMET-128 to use multiple weak keys. After solving this system of equations, the adversary request the decryption query (C', T) , where $C' = C_x \parallel C_3 \parallel \dots \parallel C_m$. If Z_0 is known to be a weak key, the attack succeeds after a single query with probability 1. Given the probability of Z_0 being a weak key is 2^{-64} the overall complexity of the attack is 2^{64} online queries, 0 offline queries and succeeds with probability close to 1.

6.3.3. Key Recovery Attack

The previous existential forgery attack can be used as a filter to discover the occurrence of a weak key. Once the forgery succeeds, we know that Z during the message encryption phase of the algorithm has one of the weak key values, which are 2^{64} values. The attacker can then choose a message that has been previously encrypted with a weak key, and reverse the algorithm with each of these values. Since the master key K is used as an IV in COMET-128, this will lead to 2^{64} possible key candidates. For each of these key candidates, the attacker can apply $\text{KDF}(N, K)$ and verify whether the KDF generates the corresponding Z . Since the probability that $E_K(N) = Z$ is 2^{-128} , we expect to be able to uniquely identify the master key at this point, which completely breaks the system. The complexity is 2^{65} offline block cipher queries. This attack is given in details in Figure 6.5 where COMETE and COMETD represent online queries to the encryption and decryption oracles, respectively, $\text{forge}(C)$ represents the forgery attack in Section 6.3.2, $\mathcal{Z}_{\text{weak}}$ is the set of weak keys of COMET-128 with empty AD and $\text{AESE}_K/\text{AESD}_K$ are the offline primitive queries for AES encryption and decryption, respectively. $\leftarrow^{\$}$ represents random sampling without replacement, in order to respect the nonce model, so N and M are never repeated. $\perp$ represents failed decryption. The first loop of the algorithm (lines 2-8) requires at most 5 blocks of storage, or 80 bytes, and is expected to run 2^{64} times, with 2^{65} data complexity. The second loop (lines 10-18) runs exactly 2^{64} times, with two primitive calls for each iteration, and requires 80 bytes of storage, in addition to the successful key candidates. Due to the properties of AES as a secure block cipher, it is expected that only 1 candidate is successful.

6.3.4. Attacks in the Multi-Key Setting

Combining the previous two attacks with the multi-key analysis shows a security concern about the design of COMET-128. While this flaw does not violate the 64-bit security and NIST requirements in the single-key setting, in practice it can lead to potential issues. Similar to COFB and HyENA, given μ users, the per-user forgery security is reduced to $64 - \log(\mu)$ bits. For example, given 4 million users, forgery is successful against at least 1 user with close to 1 probability given 2^{40} queries per user. Once the forgery succeeded against at least 1 user, the corresponding key can be recovered using 2^{65} offline encryptions. Recovering as many as one master key using 2^{40} queries per user and 2^{65} offline queries cannot be considered impractical, as it is well within the practical limits set by the NIST for each individual user and the attacker can still use this key $2^{50} - 2^{40}$ times to

6. Analysis of the Rekey-and-Chain Schemes

```

1.   $M_x \leftarrow \perp$ 
2.  while  $M_x = \perp$  do
3.     $M \xleftarrow{\$} \{0, 1\}^{2n}$ 
4.     $N \xleftarrow{\$} \{0, 1\}^n$ 
5.     $(C, T) \leftarrow \text{COMETE}(M, N)$ 
6.     $C_x \leftarrow \text{forge}(C)$ 
7.     $M_x \leftarrow \text{COMETD}(C_x, T, N)$ 
8.  endwhile
9.   $i = 0$ 
10. for  $Z \in \mathcal{Z}_{\text{weak}}$  do
11.   $S \leftarrow \text{Shuffle}^{-1}(M_x \oplus C_x)$ 
12.   $IV \leftarrow \text{AESD}_Z(S)$ 
13.   $Y \leftarrow \text{AESE}_{IV}(N)$ 
14.  if  $Y = Z$  do
15.     $K_i \leftarrow IV$ 
16.     $i = i + 1$ 
17.  endif
18. endfor
19. return  $\{K_j | 0 \leq j < i\}$ 

```

Figure 6.5.: Full key recovery attack in the single-key setting

impersonate the victim user or to eavesdrop on the communications. This attack is given in details in Figure 6.6. The notation is the same as Figure 6.5, except U is the identity of the victim user, S_μ is the set of targeted users such that $\mu = |S_\mu|$ and $\text{COMETE}_u/\text{COMETD}_u$ are online queries for a specific user u . The offline resources are the same as the single-key attack, while the online queries per user are $\mathcal{O}(\frac{2^{64}}{\mu})$.

6.3.5. Possible Fixes

We propose two simple fixes that have the potential of eliminating the problems we discussed:

1. Eliminate the weak keys completely by replacing the doubling with an arithmetic counter. However, arithmetic counters are more costly in hardware implementations. A 64-bit arithmetic counter can be very slow.
2. Use doubling over a larger field, *e.g.* $\text{GF}(2^{128})$, in order to reduce the number and probability of weak keys.

6.4. Application to mixFeed

mixFeed [121] is an AES-based AEAD algorithm submitted to round 2 of the NIST lightweight cryptography standardization process. It uses a hybrid feedback

```

1.   $M_x \leftarrow \perp$ 
2.  while  $M_x = \perp$  do
3.    for  $u \in S_\mu$  do
4.      if  $M_x = \perp$  do
5.         $M \xleftarrow{\$} \{0, 1\}^{2n}$ 
6.         $N \xleftarrow{\$} \{0, 1\}^n$ 
7.         $(C, T) \leftarrow \text{COMETE}_u(M, N)$ 
8.         $C_x \leftarrow \text{forge}(C)$ 
9.         $M_x \leftarrow \text{COMETD}_u(C_x, T, N)$ 
10.       if  $M_x \neq \perp$  then  $U \leftarrow u$ 
11.     endif
12.   endfor
13. endwhile
14.   $i = 0$ 
15.  for  $Z \in \mathcal{Z}_{\text{weak}}$  do
16.     $S \leftarrow \text{Shuffle}^{-1}(M_x \oplus C_x)$ 
17.     $IV \leftarrow \text{AESD}_Z(S)$ 
18.     $Y \leftarrow \text{AESE}_{IV}(N)$ 
19.    if  $Y = Z$  do
20.       $K_i \leftarrow IV$ 
21.       $i = i + 1$ 
22.    endif
23.  endfor
24.  return  $(U, \{K_j | 0 \leq j < i\})$ 

```

Figure 6.6.: Full key recovery attack in the multi-key setting

structure, where half the input to the block cipher comes directly from the plaintext, while the other half is generated from the previous block cipher call and the plaintext in a CBC-like manner. The initial session key is generated using a KDF that depends on the master key K and the nonce N . Then each block key is the output of applying a permutation Perm to the previous block key. The permutation Perm is defined as 11 rounds of the AES key scheduling Algorithm [17]. A message M of length that is multiple of n , is parsed into half blocks $(M_1, M_2, \dots, M_m) \xleftarrow{\frac{n}{2}} M$. The encryption part Enc is shown in Figure 6.7. γ_M is a constant that is used to distinguish complete and incomplete blocks. It is irrelevant to our analysis.

6.4.1. Weak Key Analysis of mixFeed

The designers of mixFeed discuss the multi-key analysis in a brief statement in the specifications. However, they do not mention the weak key analysis. At first, it is not obvious why the weak key analysis is relevant to mixFeed. However, when we

6. Analysis of the Rekey-and-Chain Schemes

Figure 6.7.: Part of the encryption part Enc of mixFeed

Table 6.3.: 8 round unrolling of the AES key schedule

Round 0	W_0	W_1	W_2	W_3
Round 1	$W_0 \oplus f_0$	$W_1 \oplus W_0 \oplus f_0$	$W_2 \oplus W_1 \oplus W_0 \oplus f_0$	$W_3 \oplus W_2 \oplus W_1 \oplus W_0 \oplus f_0$
Round 2	$W_0 \oplus f_0 \oplus f_1$	$W_1 \oplus f_1$	$W_2 \oplus W_0 \oplus f_0 \oplus f_1$	$W_3 \oplus W_1 \oplus f_1$
Round 3	$W_0 \oplus f_0 \oplus f_1 \oplus f_2$	$W_1 \oplus W_0 \oplus f_0 \oplus f_2$	$W_2 \oplus W_1 \oplus f_1 \oplus f_2$	$W_3 \oplus W_2 \oplus f_2$
Round 4	$W_0 \oplus f_0 \oplus f_1 \oplus f_2 \oplus f_3$	$W_1 \oplus f_1 \oplus f_3$	$W_2 \oplus f_2 \oplus f_3$	$W_3 \oplus f_3$
Round 8	$W_0 \oplus \bigoplus_{i=0}^7 f_i$	$W_1 \oplus \bigoplus_{i=0}^3 f_{2+i+1}$	$W_2 \oplus f_2 \oplus f_3 \oplus f_6 \oplus f_7$	$W_3 \oplus f_3 \oplus f_7$

study how the mode operates, it is quite similar to modes like COMET-128, except that the key update function between blocks is not a multiplication by constant over a finite field, but it is the key schedule permutation of AES itself. In other words, every block cipher call takes as a key $Z_i = \text{Perm}(Z_{i-1})$, where Z_{i-1} is the key used in the previous block and Perm is the permutation that applies 11 rounds of the AES key schedule. As explained in Section 6.2.2, different types of forgery succeed if the key is repeated, *i.e.*, if the permutation cycle used to update the key is smaller than the message length. If the permutation is well designed, *e.g.* maximal length LFSR or arithmetic counter, the probability of this event should be very low. Also, if the permutation is an ideal permutation picked uniformly at random, it should have n cycles whose lengths follow a Poisson distribution.

The AES key schedule permutation is not designed to be an ideal permutation and it should not be used as one. It can be described as a permutation over four 32-bit words, which consists of 11 rounds. The rounds differ only in the round constant. We define the permutation f_c over a 32 bit word as the feedback function in round c as:

$$W \rightarrow \text{SubWord}(W \ggg 8) + \text{rcon}(c)$$

where $W \ggg r$ represents bitwise right rotation of W by r bits and $\text{rcon}(c)$ is defined as $x^{\text{cm}od 11} | 0^{24}$ such that x is defined over $\text{GF}(2^8)$. Given this permutation, a single round of the AES key schedule can be defined as

$$W_0, W_1, W_2, W_3 \rightarrow W_0 \oplus f_c, W_1 \oplus W_0 \oplus f_c, W_2 \oplus W_1 \oplus W_0 \oplus f_c, W_3 \oplus W_2 \oplus W_1 \oplus W_0 \oplus f_c$$

where f_c is applied to W_3 and eight unrolled rounds can be defined as in Table 6.3.

In fact, there is an iterative structure over 4 rounds, where we can write the value of any key word after 4 rounds in terms of the initial value of this word and a certain set of feedback functions. If a key is a fixed point over R rounds, where R is a multiple of 4, then the involved feedback functions must add up to 0. If the feedback function is ideal, we expect this to happen with probability 2^{-32} for each word and 2^{-128} in total. Given the structure of the feedback function, this is not

the case. Solving the equations, we can see that there is only 1 value which is a fixed point after 4 rounds, and in general the conditions for a fixed point after R rounds are

$$\begin{aligned}
& \bigoplus_{i=0}^{R/4-1} f_{3+4i} = 0 \\
& \bigoplus_{i=0}^{R/4-1} f_{2+4i} = 0 \\
& \bigoplus_{i=0}^{R/4-1} f_{1+4i} = 0 \\
& \bigoplus_{i=0}^{R/4-1} f_{4i} = 0
\end{aligned}$$

For example, fixed points over 8 rounds must satisfy

$$\begin{aligned}
f_3 \oplus f_7 &= 0 \\
f_2 \oplus f_6 &= 0 \\
f_1 \oplus f_5 &= 0 \\
f_0 \oplus f_4 &= 0
\end{aligned}$$

The last condition can be written as $f_0(W_3) \oplus f_4(W_3 \oplus f_3) = 0$. Since f_0 and f_4 differ only in the constant value, we can rewrite the condition as

$$\text{SubWord}(W \ggg 8) \oplus \text{SubWord}(W \ggg 8 \oplus \Delta) = \delta$$

where $\Delta = f_3$ and $\delta = \text{rcon}(4) \oplus \text{rcon}(0)$. Clearly, this is a non-linear equation. What is interesting, is that this equation is defined over the Sbox of AES and can be divided into three equations on the form $\text{Sbox}(x) \oplus \text{Sbox}(x \oplus y) = 0$ and one equation of the form $\text{Sbox}(x) \oplus \text{Sbox}(x \oplus y) = a$. For the first three equations, $y = 0$ since the AES Sbox is bijective, while for the last one y has 127 possible values that can be retrieved from the Difference Distribution Table (DDT) of the AES Sbox. Hence, we reduce the possibilities of f_3 to 127 values. Then,

$$f_3(W_2 \oplus W_3 \oplus f_2) \oplus f_7(W_2 \oplus W_3 \oplus f_2 \oplus f_6) = f_3(W_2 \oplus W_3 \oplus f_2) \oplus f_7(W_2 \oplus W_3) = 0$$

Hence, similar arguments can be made about f_2 and similarly f_1 and f_0 . By such argument, one expects roughly about $2^{27.9}$ fixed points for the reduced-round AES key schedule of 8 rounds. One can go analyzing more rounds. While this problem is interesting on its own regard, it is not within the scope of our result and we leave it to future work. We just mention the analysis to show that the AES Key Schedule is far from an ideal permutation and also because the cycle length we have found is a multiple of 4.

We have run a simple cycle finding script using brute force and found at least

6. Analysis of the Rekey-and-Chain Schemes

Table 6.4.: Representatives of 20 Cycles of length= 14018661024 for the AES Key Schedule 11 Round Permutation Used in mixFeed

000102030405060708090a0b0c0d0e0f
00020406080a0c0e10121416181a1c1e
0004080c1014181c2024282c3034383c
00081018202830384048505860687078
00102030405060708090a0b0c0d0e0f0
101112131415161718191a1b1c1d1e1f
20222426282a2c2e30323436383a3c3e
4044484c5054585c6064686c7074787c
80889098a0a8b0b8c0c8d0d8e0e8f0f8
303132333435363738393a3b3c3d3e3f
707172737475767778797a7b7c7d7e7f
000306090c0f1215181b1e2124272a2d
00050a0f14191e23282d32373c41464b
00070e151c232a31383f464d545b6269
000d1a2734414e5b6875828f9ca9b6c3
00152a3f54697e93a8bdd2e7fc11263b
00172e455c738aa1b8cfe6fd142b4259
00183048607890a8c0d8f00820385068
001c3854708ca8c4e0fc1834506c88a4
001f3e5d7c9bbad9f81736557493b2d1

20 cycles of length $14018661024 \sim 2^{33.71}$ ¹, out of 33 seeds we have tried. We give a representative of each of those cycles in Table 6.4, in case the reader wants to verify the results. This means that we have identified at least $2^{38.02}$ weak keys which allow forgery of messages of length $2^{33.71} + 1$ blocks into messages of one block. Finding each of these cycles takes around 1 hour on a single-core personal computer. While our findings are good enough to show a gap in the analysis of mixFeed, in a follow-up Leurent and Pernot managed to implement our attack proposed in Section 6.2.2 with $l = 14018661024$, data complexity of ~ 220 GB and success probability of $\sim 45\%$, showing that our attack against mixFeed is indeed a practical break.

Similar to COMET-128, COFB and HyENA, the forgery attack against mixFeed is vulnerable to multi-key security degradation. It is worth noting that, unlike in the case of COMET-128, the weak key analysis does not lead to the master key recovery, since mixFeed does not use the master key during the main part of the encryption.

¹In the original paper, appearing in ToSC 2019 (4) [116], the cycle length was reported as 1133759136. However, it turned out there was an overflow bug in the cycle finding script and the correct length is 14018661024. This bug was identified by Leurent and Pernot [124].

6.5. Misuse in RaC Schemes: Attack on mixFeed

Nonce-misuse scenarios are important for the practical security of some applications. This raises the question of whether the RaC schemes are secure in such model. The designers of mixFeed claim that there is no conventional privacy security in case of nonce misuse. However, they initially conjectured that integrity security remains until 2^{32} data in that situation. Our analysis shows that this claim may only be true for the case where the plaintext size is smaller than 16 bytes, which is a very restrictive scenario. In this section, we show a simple forgery attack that requires only 32 bytes of plaintext and succeeds with probability 1 after only a single nonce repetition.

Attack on the mixFeed AEAD mode in the nonce-misuse model

1. The adversary generates an associated data string A of arbitrary length and a plaintext string M of 32 bytes, divided into 4 words of 8 bytes each: M_0, M_1, M_2, M_3 .
2. He/she generates a plaintext string M' of 32 bytes, divided into 4 words of 8 bytes each: M'_0, M'_1, M'_2, M'_3 .
3. He/she sends the following two queries to the encryption oracle:
 - a) (N, A, M) the ciphertext/tag pair (C, T) , where C consists of 4 words of 8 bytes each.
 - b) (N, A, M') and receives the ciphertext/tag pair (C', T') , where C' consists of 4 words of 8 bytes each.
4. The adversary generates a ciphertext string $C'' = C_0 \parallel C_1 \parallel C_2 \oplus M_2 \oplus M'_2 \parallel C'_3$.
5. He/she sends following challenge query to the decryption oracle: (N, A, C'', T') . The decryption succeed with probability 1.

Attack Details In order to understand why the attack works, we trace the intermediate values in the targeted parts of the execution for the encryption and decryption queries. In Figures 6.8 and 6.9, we depict the encryption calls for M and M' . The goal of the attacker is to match the chaining values at the input of the second encryption in the challenge query. Due to the hybrid feedback structure, different strategies need to be used for different words of the ciphertext. For the ciphertext feedback branch (bottom branch of Figure 6.10), we simply change C_3 to C'_3 , which directly decides the input to the block cipher in the decryption process. For the plaintext feedback branch (top branch of Figure 6.10), using $C''_2 = C_2 \oplus M_2 \oplus M'_2$ as the ciphertext word leads M'_2 at the input of the block cipher, since $C_2 \oplus M_2$ is the output of the block cipher in the previous call (Figure 6.8). Hence, the second encryption call matches the second encryption call from Figure 6.9. Since all the calls before this call match Figure 6.8 and all the calls afterwards match Figure 6.9, using the same Tag T' from Figure 6.9 leads to a successful forgery attack.

6. Analysis of the Rekey-and-Chain Schemes

Figure 6.8.: Trace of the first encryption query

Figure 6.9.: Trace of the second encryption query

Figure 6.10.: Trace of the challenge decryption query

6.5. Misuse in RaC Schemes: Attack on mixFeed

Example We have verified our attack using the reference implementation of mixFeed [121]. We generated the example forgery shown below.

The two encryption queries are:

```
Query 1
K=000102030405060708090A0B0C0D0E0F
N=000102030405060708090A0B0C0D0E
M=000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F
A=000102030405060708090A0B0C0D0E0F
C=F4C757EEC527CAF2083A4E0E3548EB4683EA28AB2C68D70AA9A90EF42CA6451E
T=324946C94446C53C5C77E661FCE80750

Query 2
K=000102030405060708090A0B0C0D0E0F
N=000102030405060708090A0B0C0D0E
M'=0008101820283038404850586068707880889098A0A8B0B8C0C8D0D8E0E8F0F8
A=000102030405060708090A0B0C0D0E0F
C'=F4CE45F5E10AFCCD407B145D592D95314E21C4BB0B694B376CC43C361BA8B89A
T'=2C55A84A127C07C611B2E35175B7E28C
```

and the challenge ciphertext is

```
C"=F4C757EEC527CAF2083A4E0E3548EB464E21C4BB0B694B377178C437D053ABF9
T'=2C55A84A127C07C611B2E35175B7E28C
```

where the decryption oracle outputs

```
M"=000102030405060708090A0B0C0D0E0FDDDAFE0333148A2AC0C8D0D8E0E8F0F8
```

Instantiating the attack with different Associated data strings

The attack can also be instantiated using only 16 bytes of plaintext, where the encryption queries have different associated data strings of equal number of bytes. We have verified this instance of the attack using the reference implementation of mixFeed [121]. We generated the example forgery shown below.

```
Query 1
K=000102030405060708090A0B0C0D0E0F
N=000102030405060708090A0B0C0D0E
M=000102030405060708090A0B0C0D0E0F
A=000102030405060708090A0B0C0D0E0F
C=F4C757EEC527CAF2083A4E0E3548EB46
T=89E7DB42C6777B7BBAFE1ABB4022AF28

Query 2
K=000102030405060708090A0B0C0D0E0F
N=000102030405060708090A0B0C0D0E
M'=00081018202830384048505860687078
A'=00081018202830384048505860687078
C'=BCBA409676B0679FB27F7F70D1A0A6D9
T'=84AE15E2E3347E8886E59A759E43A0D9
```

and the challenge ciphertext is

6. Analysis of the Rekey-and-Chain Schemes

```
C"=BCBA409676B0679F407B145D592D9531  
T'=84AE15E2E3347E8886E59A759E43A0D9
```

where the decryption oracle outputs

```
M"=487C157BB792AB6A4048505860687078
```

7. Lightweight AEAD from Tweakable Block Ciphers

In this chapter, we present two new TBC-based AEAD operating modes, named Romulus and Remus. They are lightweight, very efficient, and highly-secure NAE (Romulus-N and Remus-N) and MRAE (Romulus-M and Remus-M) schemes. The designs in this chapter are published in *IACR Transactions on Symmetric Cryptology* (2020) [118] and are the basis of 3 submissions to the NIST lightweight cryptography standardization project: Romulus [112], Remus [114] and TGIF [101]. Romulus is currently a second-round candidate in the same project/competition.

The Romulus Mode. Our first proposal is Romulus. Its NAE version Romulus-N requires fewer TBC calls than Θ CB3 thanks to the faster MAC computation for associated data, while the hardware implementation is significantly smaller than Θ CB3 thanks to the reduced state size and inverse-freeness (i.e., TBC inverse is not needed). In fact, thanks to a careful integration of the TBC inside the mode, Romulus-N’s state size is comparable to what is needed for computing the TBC alone. Moreover, it encrypts an n -bit plaintext block with just one call of the n -bit block TBC, hence there is no efficiency loss. Romulus-N is extremely efficient for small messages, which is particularly important in many lightweight applications, requiring for example only 2 TBC calls to handle one associated data block and one message block (in comparison, other designs like Θ CB3, OCB3, TAE, CCM require from 3 to 5 (T)BC calls in the same situation).

Romulus achieves all these advantages without any security penalty, i.e., Romulus guarantees full n -bit security in the nonce-respecting model, which is a similar security bound to Θ CB3. In addition, the n -bit security of Romulus is proved under the standard model, which provides a high-level assurance for security not only quantitatively but also qualitatively. To elaborate a bit more, with a security proof in the standard model, one can precisely connect the security status of the primitive to the overall security of the mode that uses this primitive. In our case, for each of the members of Romulus, the best attack on it implies a chosen-plaintext attack (CPA) in the single-key setting against the underlying TBC, i.e., unless the TBC is broken by CPA adversaries in the single-key setting, Romulus indeed maintains the claimed n -bit security. Such a guarantee is not possible with non-standard models and it is often not easy to deduce the impact of a found “flaw” of the primitive to the security of the mode. In a more general context, this gap between the proof and the actual security is best exemplified by “uninstantiable” Random Oracle-Model schemes [128]. To evaluate the security of Romulus, with the standard model proof, we can focus on the security evaluation of the TBC, while this type of focus is not possible in schemes with proofs in non-standard models.

An interesting feature of Romulus is that it can reduce area depending on the use cases, without harming security. If it is enough to have a relatively short nonce or

7. Lightweight AEAD from Tweakable Block Ciphers

a short counter (or both), which is common to low-power networks, we can directly save area by truncating the corresponding tweak length. This is possible if the internal TBC allows to reduce area if a part of its tweak is never used. A member of Romulus (Romulus-N2) particularly benefits from this feature. Note that this type of area reduction is not possible with conventional permutation/sponge-based AE schemes: it only offers a throughput/security trade-off, *i.e.*, from the fixed permutation, one can adapt the rate to trade throughput and security.

The Remus Mode. Our second proposal is Remus. It shares its basic structure with Romulus and therefore also inherits its overall implementation advantages. The biggest difference of Remus from Romulus is the way it instantiates a TBC. Specifically, Remus takes an approach of utilizing the whole tweakkey state [20] of the TBC as a function of the key and tweak, using a tweak-dependent key derivation. In contrast to this, in Romulus, the TBC is used in the standard keying setting (*i.e.*, tweakkey state takes a persistent key material and a changing tweak). The tweak-dependent key derivation allows us to use a smaller variant of the TBC than those used by Romulus, and brings us better efficiency and comparable bit security. The downside is that the security proof of Remus is not based on the standard assumption of its cryptographic core (namely, the fact that TBC can be modelled as a tweakable pseudorandom permutation) as done for Romulus. Instead, we can prove the security of Remus by assuming the TBC as an ideal cipher (thus ideal cipher model proof), or, assuming the pseudorandomness of another TBC built on top of the standard BC, called ICE. The latter is a standard model proof but the assumption is still different from the pseudorandomness assumption of the TBC. This means that Remus is not a simple optimization of Romulus but is the consequence of trade-off between (qualitative) security and efficiency.

We specify a set of members for Remus that have different TBC (ICE) instantiations based on a block cipher in order to provide security-efficiency trade-offs. As with Romulus, the overall structure of Remus is partially similar to a TBC-variant of the block cipher mode COFB [46] yet, we make numerous refinements to achieve our design goal. Consequently, as a mode of TBC ICE, the NAE variant Remus-N achieves a significantly smaller state size than ΘCB3 [10], the typical choice for TBC-based AE mode, while keeping the equivalent efficiency (*i.e.*, the same number of TBC calls). Also Remus is inverse-free (*i.e.*, no TBC decryption routine is needed) unlike ΘCB3 . For security, it allows either classical $n/2$ -bit security or full n -bit security depending on the variant of ICE.

To see the superior performance of Remus-N, let us compare n -bit secure Remus-N2 with other n -bit secure AE schemes that process n bits of data, such as conventional sponge/permutation-based AEs using a $3n$ -bit permutation with n -bit rate which would provide n -bit security. Both have $3n$ state bits and process an n -bit message block per primitive call. However, the internal cryptographic primitive for Remus-N2, which is an n -bit TBC with n -bit tweakkey, is expected to be much more lightweight and/or faster because of smaller output size ($3n$ vs n bits). Since its tweakkey is n bits, it is even smaller than the members of Romulus, which are n -bit secure while using tweakkey state of $2n$ or $3n$ bits. Both sponge/permutation-based schemes and Remus rely on non-standard models (random permutation or ideal cipher).

An additional feature of Remus is that it offers a very flexible security/size trade-off without changing the throughput. In more detail, Remus contains $n/2$ -bit

secure variants (Remus-N1 and Remus-M1) and n -bit secure variants (Remus-N2 and Remus-M2). Their difference is only in the existence of the second (block) mask, which increases the state size. If the latter is too big and n -bit security is considered an overkill, it is possible to derive an intermediate variant by truncating the second mask to (say) $n/2$ bits. It will be $(n + n/2)/2 = 3n/4$ -bit secure. For simplicity, we did not include such variants in the official members of Remus, however, this flexibility would be useful in practice.

Misuse Resistance. Romulus-M and Remus-M are the MRAE versions of Romulus and Remus respectively, and they follow the general SIV construction [44]. However, they reuse the components of Romulus-N and Remus-N as much as possible, simply obtained by processing the message twice by Romulus-N or Remus-N. This allows a faster and smaller scheme than TBC-based MRAE SCT [11], yet, we maintain the strong security features of SCT. That is, Romulus-M achieves n -bit security against nonce-respecting adversaries and $n/2$ -bit security against nonce-misusing adversaries, while a variant of Remus-M achieves the same security as the corresponding Remus-N variant against nonce-respecting adversaries and $n/2$ -bit security against nonce-misusing adversaries. Moreover, Romulus-M and Remus-M enjoy a very useful security feature called graceful degradation introduced in [11]. This ensures that the full security is almost retained if the number of nonce repetitions during encryption is limited (which is the main targeted scenario in practice). Thanks to the shared components, most of the advantages of Romulus-N and Remus-N mentioned above also hold for Romulus-M and Remus-M.

Instantiations with Skinny. As the underlying TBC, we adopt Skinny proposed at CRYPTO 2016 [18], leading to the two submissions Romulus [112] and Remus [114] for the NIST LWC standardization competition. See Section 7.7 for the details of the instantiation.

7.1. Specifications

7.1.1. Padding

For $X \in \{0, 1\}^{\leq l}$ of length multiple of 8 (i.e., byte string), let

$$\text{pad}_l(X) = \begin{cases} X & \text{if } |X| = l, \\ X \parallel 0^{l-|X|-8} \parallel \text{len}_8(X) & \text{if } 0 \leq |X| < l, \end{cases}$$

where $\text{len}_8(X)$ denotes the one-byte encoding of the byte-length of X (we assume that $l < 256$ bytes). Here, $\text{pad}_l(\varepsilon) = 0^l$. For example, when $l = 128$, $\text{len}_8(X)$ has 16 variations (i.e., byte length 0 to 15), and we encode it to the last 4 bits of $\text{len}_8(X)$ (for example, $\text{len}_8(11) = 00001011$). The case $l = 64$ is similarly treated, by using the last 3 bits.

7.1.2. State Update Function

In this section, we specify our two new authenticated encryption modes, Romulus and Remus. First, we describe the state update function, that is common to both designs. Let G be an $n \times n$ binary matrix defined as an $n/8 \times n/8$ diagonal matrix of 8×8 binary sub-matrices G_s . They are defined as

7. Lightweight AEAD from Tweakable Block Ciphers

$$G = \begin{pmatrix} G_s & 0 & 0 & \dots & 0 \\ 0 & G_s & 0 & \dots & 0 \\ \vdots & & \ddots & & \vdots \\ 0 & \dots & 0 & G_s & 0 \\ 0 & \dots & 0 & 0 & G_s \end{pmatrix}, \quad G_s = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}, \quad (7.1) \quad (7.2)$$

where 0 for G represents the 8×8 zero matrix.

The state update function $\rho : \{0, 1\}^n \times \{0, 1\}^n \rightarrow \{0, 1\}^n \times \{0, 1\}^n$ and its inverse $\rho^{-1} : \{0, 1\}^n \times \{0, 1\}^n \rightarrow \{0, 1\}^n \times \{0, 1\}^n$ are defined as

$$\rho(S, M) = (S', C), \quad (7.3)$$

where $C = M \oplus G(S)$ and $S' = S \oplus M$. Similarly,

$$\rho^{-1}(S, C) = (S', M), \quad (7.4)$$

where $M = C \oplus G(S)$ and $S' = S \oplus M$. We note that we abuse the notation by writing ρ^{-1} as this function is only the inverse of ρ according to its second parameter. For any $(S, M) \in \{0, 1\}^n \times \{0, 1\}^n$, if $\rho(S, M) = (S', C)$ holds then $\rho^{-1}(S, C) = (S', M)$. Besides, we remark that $\rho(S, 0^n) = (S, G(S))$ holds.

Note that other state update functions are possible, but need to fulfill the following security condition. First, let $G^{(i)}$ be an $n \times n$ matrix that is equal to G except the $(i + 1)$ -st to n -th rows, which are set to all zero. Here, $G^{(0)}$ is the zero matrix and $G^{(n)} = G$, and for $X \in \{0, 1\}^n$, $G^{(i)}(X) = \text{lm}_i(G(X)) \parallel 0^{n-i}$ for all $i = 0, 8, 16, \dots, n$; note that all variables are byte strings, and $\text{lm}_i(X)$ are the leftmost $i/8$ bytes. Let I denote the $n \times n$ identity matrix.

Definition 7.1

We say that the $n \times n$ binary matrix G is *sound* if (1) G is regular (full-rank) and (2) $G^{(i)} + I$ is regular for all $i = 8, 16, \dots, n$.

We have verified the soundness of our G proposal, for a range of n including $n = 64$ and $n = 128$, by a computer program.

Extension to Bit Strings. When the inputs are arbitrary bit strings rather than byte strings, we need $\text{pad}_\ell(X)$ to be injective on all $X \in \{0, 1\}^{<\ell}$ and keep $\text{pad}_\ell(X) = X$ when $|X| = \ell$ as before. For example we can use the popular 10^* padding. We also need to extend Definition 7.1 so that the second condition is changed to require $G^{(i)} + I$ to be regular for all $i = 1, \dots, n$. In fact, our matrix G fulfills this for $n = 64, 128$.

7.1.3. Parameters

We use the following notation for the parameters of **Remus** and **Romulus**: nonce length nl , key length k , message block length n , AD block length $n + t$ for **Romulus**

or n for Remus, counter bit length d , tag length τ . While we fix $\tau = n$, a tag for NAE schemes can be truncated if needed, at the cost of decreased security against forgery (see Section 7.2). We provide in Section 7.7 actual values of these parameters for our concrete proposed instances.

Remus uses as internal primitive a block cipher $E : \mathcal{K} \times \mathcal{M} \rightarrow \mathcal{M}$ with $\mathcal{M} = \{0, 1\}^n$ and $\mathcal{K} = \{0, 1\}^k$ and a mode ICmode to convert it into a TBC, ICmode $\in \{\text{ICE1}, \text{ICE2}, \text{ICE3}\}$ (for Ideal Cipher Encryption). We use the term ICE to denote this family of TBCs. Each TBC is a mapping $\mathcal{K} \times \mathcal{T} \times \mathcal{M} \rightarrow \mathcal{M}$, where $\mathcal{T} = \mathcal{N} \times \mathcal{D} \times \mathcal{B}$ is the tweak space with nonce space $\mathcal{N} = \{0, 1\}^{nl}$, counter space $\mathcal{D} = \llbracket 2^d - 1 \rrbracket$, and $\mathcal{B} = \llbracket 256 \rrbracket_0$ for domain separation (i.e., deriving a small number of independent instances). Typically, $\mathcal{B}$ is represented as a byte.

On the other hand, Romulus uses a TBC $\tilde{E} : \mathcal{K} \times \overline{\mathcal{T}} \times \mathcal{M} \rightarrow \mathcal{M}$, where $\mathcal{K} = \{0, 1\}^k$, $\mathcal{M} = \{0, 1\}^n$, and $\overline{\mathcal{T}} = \mathcal{T} \times \mathcal{B} \times \mathcal{D}$. Here, $\mathcal{T} = \{0, 1\}^t$, $\mathcal{D} = \llbracket 2^d - 1 \rrbracket_0$, and $\mathcal{B} = \llbracket 256 \rrbracket_0$ for parameters t and d , and $\mathcal{B}$ is also represented as a byte. For tweak $\overline{T} = (T, B, D) \in \overline{\mathcal{T}}$, T is always assumed to be a byte string including ε , and t is a multiple of 8. $\mathcal{T}$ will be used to process the nonce or an AD block, $\mathcal{D}$ will be used for counter, while $\mathcal{B}$ is for domain separation. When the counter value is $i \in \mathcal{D}$, we write $\bar{i}$ to denote the i -th clocking of the counter as a part of the tweak (e.g. see Figure 7.1).

NAE and MRAE Families. Romulus and Remus each have two families, Romulus-N, Romulus-M, Remus-N and Remus-M, and each family consists of several members (the sets of parameters). The x-N families implement nonce-based AE (NAE) secure against nonce-respecting adversaries, and the x-M families implement nonce misuse-resistant AE (MRAE) introduced by Rogaway and Shrimpton [44]. The names Romulus and Remus stand for the corresponding set of two families.

7.1.4. Romulus

The specification of the NAE mode Romulus-N is shown in Figure 7.1, while Figure 7.2 gives a more graphical representation. Similarly, the specification of the MRAE mode Romulus-M is shown in Figure 7.3, while Figure 7.4 gives a more graphical representation.

To encrypt (N, A, M) under key K , in Romulus-N, we first hash $A = (A[1], \dots, A[a])$ into the state S in line 11 in Figure 7.1, where $A[1], A[3], A[5], \dots$ are injected into the state with ρ and $A[2], A[4], A[6], \dots$ are processed with the TBC. We then use nonce N to compute $S = \tilde{E}_K^{(N, w_A, \bar{a})}(S)$, that could be seen as the nonce-dependent MAC value of A . Then M is processed with ρ to generate C , where we keep using the TBC that takes N as a part of the input. The tag T is generated as $T = G(\tilde{E}_K^{(N, w_M, \bar{m})}(S))$ from the final state S after the process of M .

In Romulus-M, we first hash both A and M into the state S in line 20 or 22 in Figure 7.3, and then the tag T is computed as $T = G(\tilde{E}_K^{(N, w, \bar{a+m})}(S))$ in line 24. The tag generation may seem complex in appearance, but it simply follows the same process as Romulus-N to hash both A and M , and the encryption part of M is similar to Romulus-N. Note that the algorithm always assumes $t = nl$. These figures adopt the concrete values of domain separation (in the second argument of the tweak) described in Section 7.7.

7. Lightweight AEAD from Tweakable Block Ciphers

Algorithm Romulus-N.Enc_K(N, A, M) <ol style="list-style-type: none"> 1. $S \leftarrow 0^n$ 2. $(A[1], \dots, A[a]) \stackrel{n,t}{\leftarrow} A$ 3. if $a \bmod 2 = 0$ then $u \leftarrow t$ else n 4. if $A[a] < u$ then $w_A \leftarrow 26$ else 24 5. $A[a] \leftarrow \text{pad}_u(A[a])$ 6. for $i = 1$ to $\lfloor a/2 \rfloor$ 7. $(S, \eta) \leftarrow \rho(S, A[2i-1])$ 8. $S \leftarrow \tilde{E}_K^{(A[2i], 8, 2i-1)}(S)$ 9. end for 10. if $a \bmod 2 = 0$ then $V \leftarrow 0^n$ else $A[a]$ 11. $(S, \eta) \leftarrow \rho(S, V)$ 12. $S \leftarrow \tilde{E}_K^{(N, w_A, \bar{a})}(S)$ 13. $(M[1], \dots, M[m]) \stackrel{n}{\leftarrow} M$ 14. if $M[m] < n$ then $w_M \leftarrow 21$ else 20 15. for $i = 1$ to $m-1$ 16. $(S, C[i]) \leftarrow \rho(S, M[i])$ 17. $S \leftarrow \tilde{E}_K^{(N, 4, \bar{i})}(S)$ 18. end for 19. $M'[m] \leftarrow \text{pad}_n(M[m])$ 20. $(S, C'[m]) \leftarrow \rho(S, M'[m])$ 21. $C[m] \leftarrow \text{1mt}_{ M[m] }(C'[m])$ 22. $S \leftarrow \tilde{E}_K^{(N, w_M, \bar{m})}(S)$ 23. $(\eta, T) \leftarrow \rho(S, 0^n)$ 24. $C \leftarrow C[1] \parallel \dots \parallel C[m-1] \parallel C[m]$ 25. return (C, T) 	Algorithm Romulus-N.Dec_K(N, A, C, T) <ol style="list-style-type: none"> 1. $S \leftarrow 0^n$ 2. $(A[1], \dots, A[a]) \stackrel{n,t}{\leftarrow} A$ 3. if $a \bmod 2 = 0$ then $u \leftarrow t$ else n 4. if $A[a] < u$ then $w_A \leftarrow 26$ else 24 5. $A[a] \leftarrow \text{pad}_u(A[a])$ 6. for $i = 1$ to $\lfloor a/2 \rfloor$ 7. $(S, \eta) \leftarrow \rho(S, A[2i-1])$ 8. $S \leftarrow \tilde{E}_K^{(A[2i], 8, 2i-1)}(S)$ 9. end for 10. if $a \bmod 2 = 0$ then $V \leftarrow 0^n$ else $A[a]$ 11. $(S, \eta) \leftarrow \rho(S, V)$ 12. $S \leftarrow \tilde{E}_K^{(N, w_A, \bar{a})}(S)$ 13. $(C[1], \dots, C[m]) \stackrel{n}{\leftarrow} C$ 14. if $C[m] < n$ then $w_C \leftarrow 21$ else 20 15. for $i = 1$ to $m-1$ 16. $(S, M[i]) \leftarrow \rho^{-1}(S, C[i])$ 17. $S \leftarrow \tilde{E}_K^{(N, 4, \bar{i})}(S)$ 18. end for 19. $\tilde{S} \leftarrow (0^{ C[m] } \parallel \text{rmt}_{n- C[m] }(G(S)))$ 20. $C'[m] \leftarrow \text{pad}_n(C[m]) \oplus \tilde{S}$ 21. $(S, M'[m]) \leftarrow \rho^{-1}(S, C'[m])$ 22. $M[m] \leftarrow \text{1mt}_{ C[m] }(M'[m])$ 23. $S \leftarrow \tilde{E}_K^{(N, w_C, \bar{m})}(S)$ 24. $(\eta, T^*) \leftarrow \rho(S, 0^n)$ 25. $M \leftarrow M[1] \parallel \dots \parallel M[m-1] \parallel M[m]$ 26. if $T^* = T$ then return M else $\perp$
Algorithm $\rho(S, M)$ <ol style="list-style-type: none"> 1. $C \leftarrow M \oplus G(S)$ 2. $S' \leftarrow S \oplus M$ 3. return (S', C) 	Algorithm $\rho^{-1}(S, C)$ <ol style="list-style-type: none"> 1. $M \leftarrow C \oplus G(S)$ 2. $S' \leftarrow S \oplus M$ 3. return (S', M)

Figure 7.1.: The Romulus-N nonce-based AE mode. Lines of **[if (statement) then $X \leftarrow x$ else x']** are shorthand for **[if (statement) then $X \leftarrow x$ else $X \leftarrow x'$]**. The dummy variable η is always discarded. We use Romulus-N1 as working example. For other Romulus-N members, the values of the bits b_7 and b_6 in the domain separation (specified in Section 7.7) need to be adapted accordingly.

Figure 7.2.: The Romulus-N nonce-based AE mode. (Top) process of AD with an even number of AD blocks. (Middle) process of AD with an odd number of AD blocks. (Bottom) Encryption. We use Romulus-N1 as working example. For other Romulus-N members, the values of the bits b_7 and b_6 in the domain separation need to be adapted accordingly.

7. Lightweight AEAD from Tweakable Block Ciphers

Algorithm Romulus-M.Enc_K(N, A, M) <ol style="list-style-type: none"> 1. $S \leftarrow 0^n$ 2. $(X[1], \dots, X[a]) \xleftarrow{t} A$ 3. if $a \bmod 2 = 0$ then $u \leftarrow t$ else n 4. $(X[a+1], \dots, X[a+m]) \xleftarrow{n+t-u, u} M$ 5. if $m \bmod 2 = 0$ then $v \leftarrow u$ else $n + t - u$ 6. $w \leftarrow 48$ 7. if $X[a] < u$ then $w \leftarrow w \oplus 2$ 8. if $X[a+m] < v$ then $w \leftarrow w \oplus 1$ 9. if $a \bmod 2 = 0$ then $w \leftarrow w \oplus 8$ 10. if $m \bmod 2 = 0$ then $w \leftarrow w \oplus 4$ 11. $X[a] \leftarrow \text{pad}_u(X[a])$ 12. $X[a+m] \leftarrow \text{pad}_v(X[a+m])$ 13. $x \leftarrow 40$ 14. for $i = 1$ to $\lfloor (a+m)/2 \rfloor$ 15. $(S, \eta) \leftarrow \rho(S, X[2i-1])$ 16. if $i = \lfloor a/2 \rfloor + 1$ then $x \leftarrow x \oplus 4$ 17. $S \leftarrow \tilde{E}_K^{(X[2i], x, 2i-1)}(S)$ 18. end for 19. if $a \bmod 2 = m \bmod 2$ then 20. $(S, \eta) \leftarrow \rho(S, 0^n)$ 21. else 22. $(S, \eta) \leftarrow \rho(S, X[a+m])$ 23. $S \leftarrow \tilde{E}_K^{(N, w, a+m)}(S)$ 24. $(\eta, T) \leftarrow \rho(S, 0^n)$ 25. if $M = \epsilon$ then return (ϵ, T) 26. $S \leftarrow T$ 27. $(M[1], \dots, M[m']) \xleftarrow{n} M$ 28. $z \leftarrow M[m']$ 29. $M[m'] \leftarrow \text{pad}_n(M[m'])$ 30. for $i = 1$ to m' 31. $S \leftarrow \tilde{E}_K^{(N, 36, i-1)}(S)$ 32. $(S, C[i]) \leftarrow \rho(S, M[i])$ 33. end for 34. $C[m'] \leftarrow \text{lsb}_z(C[m'])$ 35. $C \leftarrow C[1] \parallel \dots \parallel C[m'-1] \parallel C[m']$ 36. return (C, T) 	Algorithm Romulus-M.Dec_K(N, A, C, T) <ol style="list-style-type: none"> 1. if $C = \epsilon$ then $M \leftarrow \epsilon$ 2. else 3. $S \leftarrow T$ 4. $(C[1], \dots, C[m']) \xleftarrow{n} C$ 5. $z \leftarrow C[m']$ 6. $C[m'] \leftarrow \text{pad}_n(C[m'])$ 7. for $i = 1$ to m' 8. $S \leftarrow \tilde{E}_K^{(N, 36, i-1)}(S)$ 9. $(S, M[i]) \leftarrow \rho^{-1}(S, C[i])$ 10. end for 11. $M[m'] \leftarrow \text{lsb}_z(M[m'])$ 12. $M \leftarrow M[1] \parallel \dots \parallel M[m'-1] \parallel M[m']$ 13. $S \leftarrow 0^n$ 14. $(X[1], \dots, X[a]) \xleftarrow{t} A$ 15. if $a \bmod 2 = 0$ then $u \leftarrow t$ else n 16. $(X[a+1], \dots, X[a+m]) \xleftarrow{n+t-u, u} M$ 17. if $m \bmod 2 = 0$ then $v \leftarrow u$ else $n + t - u$ 18. $w \leftarrow 48$ 19. if $X[a] < u$ then $w \leftarrow w \oplus 2$ 20. if $X[a+m] < v$ then $w \leftarrow w \oplus 1$ 21. if $a \bmod 2 = 0$ then $w \leftarrow w \oplus 8$ 22. if $m \bmod 2 = 0$ then $w \leftarrow w \oplus 4$ 23. $X[a] \leftarrow \text{pad}_u(X[a])$ 24. $X[a+m] \leftarrow \text{pad}_v(X[a+m])$ 25. $x \leftarrow 40$ 26. for $i = 1$ to $\lfloor (a+m)/2 \rfloor$ 27. $(S, \eta) \leftarrow \rho(S, X[2i-1])$ 28. if $i = \lfloor a/2 \rfloor + 1$ then $x \leftarrow x \oplus 4$ 29. $S \leftarrow \tilde{E}_K^{(X[2i], x, 2i-1)}(S)$ 30. end for 31. if $a \bmod 2 = m \bmod 2$ then 32. $(S, \eta) \leftarrow \rho(S, 0^n)$ 33. else 34. $(S, \eta) \leftarrow \rho(S, X[a+m])$ 35. $S \leftarrow \tilde{E}_K^{(N, w, a+m)}(S)$ 36. $(\eta, T^*) \leftarrow \rho(S, 0^n)$ 37. if $T^* = T$ then return M else $\perp$
Algorithm $\rho(S, M)$ <ol style="list-style-type: none"> 1. $C \leftarrow M \oplus G(S)$ 2. $S' \leftarrow S \oplus M$ 3. return (S', C) 	Algorithm $\rho^{-1}(S, C)$ <ol style="list-style-type: none"> 1. $M \leftarrow C \oplus G(S)$ 2. $S' \leftarrow S \oplus M$ 3. return (S', M)

Figure 7.3.: The Romulus-M misuse-resistant AE mode. Lines of **[if (statement) then $X \leftarrow x$ else x']** are shorthand for **[if (statement) then $X \leftarrow x$ else $X \leftarrow x'$]**. The dummy variable η is always discarded. We use Romulus-M1 as working example. For other Romulus-M members, the values of the bits b_7 and b_6 in the domain separation (specified in Section 7.7) need to be adapted accordingly. Note that in the case of empty message, no encryption call has to be performed in the encryption part.

Figure 7.4.: The Romulus-M misuse-resistant AE mode. (Top) process of AD with an even/even, even/odd, odd/even, odd/odd number of AD blocks and M blocks, respectively. (Bottom) Encryption. We use Romulus-M1 as working example. For other Romulus-M members, the values of the bits b_7 and b_6 in the domain separation need to be adapted accordingly.

7.1.5. Remus

The specification of the nonce-based AE mode Remus-N is shown in Figure 7.6, and a more graphical representation is given in Figures 7.7, 7.8, 7.9. Similarly, the specification of the misuse-resistant AE Remus-M is shown in Figure 7.10, while a more graphical representation is given in Figures 7.11 and 7.12.

The basic procedure of Remus-N and Remus-M follows that of Romulus-N and Romulus-M, respectively.

TBC ICE for Remus As described above, Remus uses ICE which consists of three variants: ICE1, ICE2 and ICE3. Each variant consists of two main components, the key derivation function $\text{KDF} : \mathcal{K} \times \mathcal{N} \rightarrow \mathcal{L} \times \mathcal{V}$, and the “core” encryption function $\text{ICEnc} : (\mathcal{L} \times \mathcal{V}) \times (\mathcal{D} \times \mathcal{B}) \times \mathcal{M} \rightarrow \mathcal{M}$. Here, $\mathcal{L} = \mathcal{K} = \{0, 1\}^k$ and $\mathcal{V} = \mathcal{M} = \{0, 1\}^n$. The algorithm of ICEnc is shown in Figure 7.5 for all variants (and a more graphical representation is given in Figures 7.7, 7.8 and 7.9). In addition, there is a tweakkey encoding function $\text{encode} : \mathcal{L} \times \mathcal{D} \times \mathcal{B} \rightarrow \mathcal{K}$ inside ICEnc. For convenience, KDF for ICE1 may also be referred to as KDF1 (KDF2 and KDF3 are defined analogously).

An encryption with ICE is performed as follows. Given a tweak $T = (N, D, B) \in \mathcal{T}$, a key $K \in \mathcal{K}$, and a plaintext $M \in \mathcal{M}$, we first derive the nonce-dependent mask values (L, V) with $\text{KDF}(N, K) \rightarrow (L, V)$, and then ICEnc encrypts M as $\text{ICEnc}(L, V, D, B, M) \rightarrow C$, using the key of the internal E determined by $\text{encode}(L, D, B)$. Here, $\text{ICEnc}(L, V, D, B, *)$ is a permutation over $\mathcal{M}$ for any (L, V, D, B) . Each variant is then defined as follows:

1. ICE1: $n = d = k$, $nl \leq n$.
 - a) $\text{KDF}(N, K) = (L, V)$ where $L = G(E_K(N \parallel 0^{n-nl}))$, $V = 0^n$.
 - b) $\text{encode}(L, D, B) = 2^D L \oplus B$, where multiplication is over $\text{GF}(2^n)$.
2. ICE2: $n = d = k$, $nl \leq n$.
 - a) $\text{KDF}(N, K) = (L, V)$ where $L' = E_K(N \parallel 0^{n-nl})$, $V' = E_{K \oplus 1}(L')$, and $L = G(L')$ and $V = G(V')$.
 - b) $\text{encode}(L, D, B) = 2^D L \oplus B$, where multiplication is over $\text{GF}(2^n)$.
3. ICE3: $nl \leq k - 8$, $d = k - 8$, n is arbitrary (however, it matters for the security of the whole Remus).
 - a) $\text{KDF}(N, K) = (L, V)$ where $L \leftarrow (N \parallel 0^{k-nl}) \oplus K$, $V \leftarrow 0^n$.
 - b) $\text{encode}(L, D, B) = (2^D L[1]) \parallel (B \oplus L[2])$, where $(L[1], L[2]) \xleftarrow{k-8} L$ and the multiplication is over $\text{GF}(2^{k-8})$ and applied to $L[1]$. Note that $|L[1]| = k - 8$ and $|L[2]| = 8$.

ICE1 and ICE2 differ only in the second mask V derived by their KDFs.

When ICE is working inside Remus, the corresponding KDF is performed only once as an initialization. For ICE1 or ICE2, KDF involves one or two calls of E and matrix multiplications by G (see above). For ICE3, KDF is just a linear operation of (N, K) . For each input block, ICE applies doubling to the derived mask values. Since doubling is a sequential operation, computing $\text{ICEnc}_{L,V}^{D+1,B}(M)$ after $\text{ICEnc}_{L,V}^{D,B'}(M')$ is easy and does not need any additional memory.

As with Romulus, the figures adopt the concrete domain separation values from Section 7.7 and use $\bar{i}$ to denote the counter value i .

Algorithm $\text{ICEnc}_{L,V}^{D,B}(M)$

1. $S \leftarrow 2^D V \oplus M$
 2. $T_K \leftarrow \text{encode}(L, D, B)$
 3. $S \leftarrow E_{T_K}(S)$
 4. $C \leftarrow 2^D V \oplus S$
 5. **return** C
-

Figure 7.5.: Definition of ICEnc , the core encryption routine of ICE. ICEnc is common to all three variants of ICE: ICE1 and ICE2 and ICE3 except the definition of encode . Note that ICE1 and ICE3 fix $V = 0^n$, hence effectively $S \leftarrow M$ (line 1) and $C \leftarrow S$ (line 4). Variables L and V are assumed to be derived from the corresponding KDF taking (N, K) , as a pre-processing. KDFs are defined in Section [7.1.5](#)

7. Lightweight AEAD from Tweakable Block Ciphers

Algorithm Remus-N.Enc _K (N, A, M) <ol style="list-style-type: none"> 1. $(L, V) \leftarrow \text{KDF}(N, K)$ 2. $S \leftarrow 0^n$ 3. $(A[1], \dots, A[a]) \xleftarrow{r} A$ 4. $(M[1], \dots, M[m]) \xleftarrow{r} M$ 5. if $A[a] < n$ then $w_A \leftarrow 13$ else 12 6. if $M[m] < n$ then $w_M \leftarrow 11$ else 10 7. $A[a] \leftarrow \text{pad}_n(A[a])$ 8. for $i = 1$ to $a - 1$ 9. $(S, \eta) \leftarrow \rho(S, A[i])$ 10. $S \leftarrow \text{ICEnc}_{L,V}^{\bar{i},4}(S)$ 11. end for 12. $(S, \eta) \leftarrow \rho(S, A[a])$ 13. $S \leftarrow \text{ICEnc}_{L,V}^{\bar{a},w_A}(S)$ 14. for $i = 1$ to $m - 1$ 15. $(S, C[i]) \leftarrow \rho(S, M[i])$ 16. $S \leftarrow \text{ICEnc}_{L,V}^{\bar{a}+\bar{i},2}(S)$ 17. end for 18. $M'[m] \leftarrow \text{pad}_n(M[m])$ 19. $(S, C'[m]) \leftarrow \rho(S, M'[m])$ 20. $S \leftarrow \text{ICEnc}_{L,V}^{\bar{a}+\bar{m},w_M}(S)$ 21. $C[m] \leftarrow \text{1mt}_{ M[m] }(C'[m])$ 22. $(\eta, T) \leftarrow \rho(S, 0^n)$ 23. $C \leftarrow C[1] \parallel C[2] \parallel \dots \parallel C[m]$ 24. return (C, T) 	Algorithm Remus-N.Dec _K (N, A, C, T) <ol style="list-style-type: none"> 1. $(L, V) \leftarrow \text{KDF}(N, K)$ 2. $S \leftarrow 0^n$ 3. $(A[1], \dots, A[a]) \xleftarrow{r} A$ 4. $(C[1], \dots, C[m]) \xleftarrow{r} C$ 5. if $A[a] < n$ then $w_A \leftarrow 13$ else 12 6. if $C[m] < n$ then $w_C \leftarrow 11$ else 10 7. $A[a] \leftarrow \text{pad}_n(A[a])$ 8. for $i = 1$ to $a - 1$ 9. $(S, \eta) \leftarrow \rho(S, A[i])$ 10. $S \leftarrow \text{ICEnc}_{L,V}^{\bar{i},4}(S)$ 11. end for 12. $(S, \eta) \leftarrow \rho(S, A[a])$ 13. $S \leftarrow \text{ICEnc}_{L,V}^{\bar{a},w_A}(S)$ 14. for $i = 1$ to $m - 1$ 15. $(S, M[i]) \leftarrow \rho^{-1}(S, C[i])$ 16. $S \leftarrow \text{ICEnc}_{L,V}^{\bar{a}+\bar{i},2}(S)$ 17. end for 18. $\tilde{S} \leftarrow (0^{ C[m] } \parallel \text{rmt}_{n- C[m] }(G(S)))$ 19. $C'[m] \leftarrow \text{pad}_n(C[m]) \oplus \tilde{S}$ 20. $(S, M'[m]) \leftarrow \rho^{-1}(S, C'[m])$ 21. $M[m] \leftarrow \text{1mt}_{ C[m] }(M'[m])$ 22. $S \leftarrow \text{ICEnc}_{L,V}^{\bar{a}+\bar{m},w_C}(S)$ 23. $(\eta, T^*) \leftarrow \rho(S, 0^n)$ 24. $M \leftarrow M[1] \parallel M[2] \parallel \dots \parallel M[m]$ 25. if $T^* = T$ then return M else $\perp$
Algorithm $\rho(S, M)$ <ol style="list-style-type: none"> 1. $C \leftarrow M \oplus G(S)$ 2. $S' \leftarrow S \oplus M$ 3. return (S', C) 	Algorithm $\rho^{-1}(S, C)$ <ol style="list-style-type: none"> 1. $M \leftarrow C \oplus G(S)$ 2. $S' \leftarrow S \oplus M$ 3. return (S', M)

Figure 7.6.: Encryption and decryption of Remus-N. It uses TBC ICE consisting of KDF and ICEnc. Lines of **[if (statement) then $X \leftarrow x$ else x']** are shorthand for **[if (statement) then $X \leftarrow x$ else $X \leftarrow x'$]**. The dummy variable η is always discarded. Remus-N1 is used as a working example. Depending on the Remus-N version, an appropriate variant of ICE will be used.

Figure 7.7.: Remus-N with ICE1 (Remus-N1). (Top) Key derivation. (Middle) Processing of AD. (Bottom) Encryption. The domain separation B being of 8 bits only, $\oplus B$ is to be interpreted as $\oplus 0^{120} \parallel B$.

7. Lightweight AEAD from Tweakable Block Ciphers

Figure 7.8.: Remus-N with ICE2 (Remus-N2). (Top) Key derivation. (Middle) Processing of AD. (Bottom) Encryption. The domain separation B being of 8 bits only, $\oplus B$ is to be interpreted as $\oplus 0^{120} \parallel B$.

Figure 7.9.: Remus-N with ICE3 (Remus-N3). (Top) Key derivation (computing $(2^D L[1]) \parallel (B \oplus L[2])$), we use the $\bar{L}$ notation to represent the fact that the multiplication is only done on $L[1]$, the first $k - 8$ bits of L). (Middle) Processing of AD. (Bottom) Encryption. The domain separation B being of 8 bits only, $\oplus B$ is to be interpreted as $\oplus 0^{120} \parallel B$.

7. Lightweight AEAD from Tweakable Block Ciphers

Algorithm Remus-M.Enc_K(N, A, M) <ol style="list-style-type: none"> 1. $(L, V) \leftarrow \text{KDF}(N, K)$ 2. $S \leftarrow 0^n$ 3. $(A[1], \dots, A[a]) \xleftarrow{r} A$ 4. $(M[1], \dots, M[m]) \xleftarrow{r} M$ 5. if $A[a] < n$ then $w_A \leftarrow 45$ else 44 6. if $M[m] < n$ then $w_M \leftarrow 47$ else 46 7. $A[a] \leftarrow \text{pad}_n(A[a])$ 8. for $i = 1$ to $a - 1$ 9. $(S, \eta) \leftarrow \rho(S, A[i])$ 10. $S \leftarrow \text{ICEnc}_{L,V}^{\bar{i},36}(S)$ 11. end for 12. $(S, \eta) \leftarrow \rho(S, A[a])$ 13. $S \leftarrow \text{ICEnc}_{L,V}^{\bar{a},w_A}(S)$ 14. for $i = 1$ to $m - 1$ 15. $(S, \eta) \leftarrow \rho(S, M[i])$ 16. $S \leftarrow \text{ICEnc}_{L,V}^{\bar{a}+\bar{i},38}(S)$ 17. end for 18. $M'[m] \leftarrow \text{pad}_n(M[m])$ 19. $(S, \eta) \leftarrow \rho(S, M'[m])$ 20. $S \leftarrow \text{ICEnc}_{L,V}^{\bar{a}+\bar{m},w_M}(S)$ 21. $(\eta, T) \leftarrow \rho(S, 0^n)$ 22. if $M = \epsilon$ then return (ϵ, T) 23. $S \leftarrow T$ 24. for $i = 1$ to $m - 1$ 25. $S \leftarrow \text{ICEnc}_{L,V}^{\bar{i}-1,34}(S)$ 26. $(S, C[i]) \leftarrow \rho(S, M[i])$ 27. end for 28. $S \leftarrow \text{ICEnc}_{L,V}^{\bar{m}-1,34}(S)$ 29. $(\eta, C'[m]) \leftarrow \rho(S, M'[m])$ 30. $C[m] \leftarrow \text{lsb}_{ M[m] }(C'[m])$ 31. $C \leftarrow C[1] \parallel C[2] \parallel \dots \parallel C[m]$ 32. return (C, T) 	Algorithm Remus-M.Dec_K(N, A, C, T) <ol style="list-style-type: none"> 1. $(L, V) \leftarrow \text{KDF}(N, K)$ 2. if $C = \epsilon$ then $M \leftarrow \epsilon$ 3. else 4. $S \leftarrow T$ 5. $(C[1], \dots, C[m]) \xleftarrow{r} C$ 6. $z \leftarrow C[m]$ 7. $C[m] \leftarrow \text{pad}_n(C[m])$ 8. for $i = 1$ to m 9. $S \leftarrow \text{ICEnc}_{L,V}^{\bar{i}-1,34}(S)$ 10. $(S, M[i]) \leftarrow \rho^{-1}(S, C[i])$ 11. end for 12. $M[m] \leftarrow \text{lsb}_z(M[m])$ 13. $M \leftarrow M[1] \parallel \dots \parallel M[m]$ 14. $S \leftarrow 0^n$ 15. $(A[1], \dots, A[a]) \xleftarrow{r} A$ 16. if $A[a] < n$ then $w_A \leftarrow 45$ else 44 17. if $M[m] < n$ then $w_M \leftarrow 47$ else 46 18. $A[a] \leftarrow \text{pad}_n(A[a])$ 19. for $i = 1$ to $a - 1$ 20. $(S, \eta) \leftarrow \rho(S, A[i])$ 21. $S \leftarrow \text{ICEnc}_{L,V}^{\bar{i},36}(S)$ 22. end for 23. $(S, \eta) \leftarrow \rho(S, A[a])$ 24. $S \leftarrow \text{ICEnc}_{L,V}^{\bar{a},w_A}(S)$ 25. for $i = 1$ to $m - 1$ 26. $(S, \eta) \leftarrow \rho(S, M[i])$ 27. $S \leftarrow \text{ICEnc}_{L,V}^{\bar{a}+\bar{i},38}(S)$ 28. end for 29. $M'[m] \leftarrow \text{pad}_n(M[m])$ 30. $(S, \eta) \leftarrow \rho(S, M'[m])$ 31. $S \leftarrow \text{ICEnc}_{L,V}^{\bar{a}+\bar{m},w_M}(S)$ 32. $(\eta, T^*) \leftarrow \rho(S, 0^n)$ 33. if $T^* = T$ then return M else $\perp$
Algorithm $\rho(S, M)$ <ol style="list-style-type: none"> 1. $C \leftarrow M \oplus G(S)$ 2. $S' \leftarrow S \oplus M$ 3. return (S', C) 	Algorithm $\rho^{-1}(S, C)$ <ol style="list-style-type: none"> 1. $M \leftarrow C \oplus G(S)$ 2. $S' \leftarrow S \oplus M$ 3. return (S', M)

Figure 7.10.: Encryption and decryption of Remus-M. Remus-M1 is used as a working example. Depending on the Remus-M version, an appropriate variant of ICE will be used.

Figure 7.11.: Remus-M with ICE1 (Remus-M1). (Top) Key derivation. (Middle-Top) Processing of AD (Middle-Bottom) Processing of M authentication (Bottom) Encryption. The domain separation B being of 8 bits only, $\oplus B$ is to be interpreted as $\oplus 0^{120} \parallel B$.

7. Lightweight AEAD from Tweakable Block Ciphers

Figure 7.12.: Remus-M with ICE2 (Remus-M2). (Top) Key derivation. (Middle-Top) Processing of AD (Middle-Bottom) Processing of M authentication (Bottom) Encryption. The domain separation B being of 8 bits only, $\oplus B$ is to be interpreted as $\oplus 0^{120} \parallel B$.

7.2. Security Analysis

In this section, we provide security bounds of Romulus and Remus. All the corresponding proofs are given in the ToSC 2020 (1) article [118]. We consider NAE notions for Romulus-N and both NAE and MRAE notions for Romulus-M, that is, both nonce-respecting (NR) and nonce-misusing (NM) adversaries.

7.2.1. Security of Romulus

Security of Romulus-N

For $A \in \{0, 1\}^*$, we say A has a AD blocks if it is parsed as $(A[1], \dots, A[a]) \xrightarrow{n,t} A$. Let $\tilde{a} = \lfloor a/2 \rfloor + 1$ which is a bound on the actual number of primitive calls for AD. Similarly, for plaintext $M \in \{0, 1\}^*$, we say M has m message blocks if $|M|_n = m$. The same applies to ciphertext C . For encryption query (N, A, M) or decryption query (N, A, C, T) of a AD blocks and m message blocks, the number of total TBC calls is at most $\tilde{a} + m$, which is called the number of *effective blocks* of a query.

While the specification assumes an n -bit tag, we extend it to be (arbitrarily) fixed truncated to $\tau \in \llbracket n \rrbracket$ bits, and show the bounds for the case of a τ -bit tag.

Security Bound 7.1

Let $\mathcal{A}$ be an NR privacy adversary against Romulus-N with time complexity t_A and with total number of effective blocks σ_{priv} . Moreover, let $\mathcal{B}$ be an NR authenticity adversary using q_d decryption queries, with total number of effective blocks for encryption and decryption queries σ_{auth} , and time complexity t_B . Then

$$\text{Adv}_{\text{Romulus-N}}^{\text{priv}}(\mathcal{A}) \leq \text{Adv}_E^{\text{tprp}}(\mathcal{A}'), \quad (7.5)$$

$$\text{Adv}_{\text{Romulus-N}}^{\text{auth}}(\mathcal{B}) \leq \text{Adv}_E^{\text{tprp}}(\mathcal{B}') + \frac{3q_d}{2^n} + \frac{2q_d}{2^\tau} \quad (7.6)$$

hold for some $(\sigma_{\text{priv}}, t_A + O(\sigma_{\text{priv}}))$ -TPRP adversary $\mathcal{A}'$, and for some $(\sigma_{\text{auth}}, t_B + O(\sigma_{\text{auth}}))$ -TPRP adversary $\mathcal{B}'$.

Security Bound 7.1 holds for all the members of Romulus-N. Although the bounds of Security Bound 7.1 look as if the adversary has almost no limitation in query complexity except q_d , there will be inherent ones from the specification, say the maximum input length. This holds for all the security claims we make.

Security of Romulus-M

For an encryption query (N, A, M) , the number of effective blocks is $\lfloor a/2 \rfloor + \lfloor m/2 \rfloor + 2 + m'$, where $(A[1], \dots, A[a]) \xrightarrow{n,t} A$, $(M[1], \dots, M[m]) \xrightarrow{n,t} M$ (or $(M[1], \dots, M[m]) \xrightarrow{t,n} M$), and $(M[1], \dots, M[m']) \xrightarrow{n} M$. For a decryption query (N, A, C, T) , it is similarly defined by

$$(C[1], \dots, C[m]) \xrightarrow{n,t} C$$

or

$$(C[1], \dots, C[m]) \xrightarrow{t,n} C, .$$

7. Lightweight AEAD from Tweakable Block Ciphers

and

$$(C[1], \dots, C[m']) \xleftarrow{n} C$$

We note that it is possible to unify Security Bounds 7.2 and 7.3 into a single statement, while we chose to differentiate them for the difference of the security model, and separating the proofs makes them more transparent.

Security Bound 7.2

Let $\mathcal{A}$ be an NR privacy adversary against Romulus-M that uses q_e encryption queries with time complexity t_A and with total number of effective blocks σ_{priv} . Let $\mathcal{B}$ be an NR authenticity adversary against Romulus-M using q_e encryption queries and q_d decryption queries, with total number of effective blocks for encryption and decryption queries σ_{auth} and with time complexity t_B . Then we have

$$\begin{aligned} \text{Adv}_{\text{Romulus-M}}^{\text{priv}}(\mathcal{A}) &\leq \text{Adv}_{\tilde{E}}^{\text{tprp}}(\mathcal{A}'), \\ \text{Adv}_{\text{Romulus-M}}^{\text{auth}}(\mathcal{B}) &\leq \text{Adv}_{\tilde{E}}^{\text{tprp}}(\mathcal{B}') + \frac{5q_d}{2^n} \end{aligned}$$

for some $(\sigma_{\text{priv}}, t_A + O(\sigma_{\text{priv}}))$ -TPRP adversary $\mathcal{A}'$, and $(\sigma_{\text{auth}}, t_B + O(\sigma_{\text{auth}}))$ -TPRP adversary $\mathcal{B}'$.

Security Bound 7.3

Let $\mathcal{A}$ be an NM privacy adversary and let $\mathcal{B}$ be an NM authenticity adversary, both are against Romulus-M, that use the same parameters as in Security Bound 7.2, and that can repeat a nonce at most $1 \leq r \leq 2^{n-1}$ times in encryption queries. Then we have

$$\begin{aligned} \text{Adv}_{\text{Romulus-M}}^{\text{nm-priv}}(\mathcal{A}) &\leq \text{Adv}_{\tilde{E}}^{\text{tprp}}(\mathcal{A}') + \frac{4r\sigma_{\text{priv}}}{2^n}, \\ \text{Adv}_{\text{Romulus-M}}^{\text{nm-auth}}(\mathcal{B}) &\leq \text{Adv}_{\tilde{E}}^{\text{tprp}}(\mathcal{B}') + \frac{4rq_e + 5rq_d}{2^n} \end{aligned}$$

for some $(\sigma_{\text{priv}}, t_A + O(\sigma_{\text{priv}}))$ -TPRP adversary $\mathcal{A}'$ and $(\sigma_{\text{auth}}, t_B + O(\sigma_{\text{auth}}))$ -TPRP adversary $\mathcal{B}'$.

Bit Security. Both Romulus-N and Romulus-M have n -bit security for NR-privacy and NR-authenticity, and Romulus-M has $n/2$ -bit security for NM-privacy and NM-authenticity, with graceful degradation in the number of nonce repetitions [11], i.e., a small number of nonce repetitions do not harm the security too much. The bounds of Romulus-N is the same as ΘCB3 except the tiny difference in their constants.

7.2.2. Security of Remus

Unlike Romulus, the security of Remus relies on the ideal cipher model, that is, the underlying $E : \mathcal{K} \times \mathcal{M} \rightarrow \mathcal{M}$ is sampled uniformly over all the block ciphers, and

can be queried at any time for both directions: $\widehat{C} \leftarrow E(\widehat{K}, \widehat{M})$ or $\widehat{M} \leftarrow E^{-1}(\widehat{K}, \widehat{C})$, where the key input $\widehat{K}$ is a part of the query. A query to E is called a primitive query. An ordinary (encryption/decryption) query to Remus-N or Remus-M may be called a construction query. The framework is similar to ICM-based TBC constructions, e.g. Mennink [129].

Security of Remus-N

For $A \in \{0, 1\}^*$, we say A has a AD blocks if $|A|_n = a$. Similarly, for plaintext $M \in \{0, 1\}^*$ we say M has m message blocks if $|M|_n = m$. The same holds for the ciphertext C . For the encryption query (N, A, M) or decryption query (N, A, C, T) of a AD blocks and m message blocks, the total number of TBC calls is at most $a + m$, which is called the number of *effective blocks* of a query. As before $\tau \in \llbracket n \rrbracket$ is the tag length.

Security Bound 7.4

Let $\mathcal{A}$ be an NR privacy adversary against Remus-N using q_e encryption (construction) queries and q_p primitive queries with total number of effective blocks in encryption queries σ_{priv} . Moreover, let $\mathcal{B}$ be an NR authenticity adversary against Remus-N using q_e encryption queries and q_d decryption queries, with total number of effective blocks for encryption and decryption queries σ_{auth} , and q_p primitive queries. Then we have

$$\text{Adv}_{\text{Remus-N1}}^{\text{priv}}(\mathcal{A}) \leq \frac{9\sigma_{\text{priv}}^2 + 4q_p\sigma_{\text{priv}}}{2^n} + \frac{2q_p}{2^n}, \quad (7.7)$$

$$\text{Adv}_{\text{Remus-N2}}^{\text{priv}}(\mathcal{A}) \leq \frac{9\sigma_{\text{priv}}^2 + 4q_p\sigma_{\text{priv}}}{2^{2n}} + \frac{2q_p}{2^n}, \quad (7.8)$$

$$\text{Adv}_{\text{Remus-N3}}^{\text{priv}}(\mathcal{A}) \leq \frac{0.5\sigma_{\text{priv}}^2}{2^{k-8}} + \frac{q_p\sigma_{\text{priv}}}{2^k}, \quad (7.9)$$

$$\text{Adv}_{\text{Remus-N1}}^{\text{auth}}(\mathcal{B}) \leq \frac{9\sigma_{\text{auth}}^2 + 4q_p\sigma_{\text{auth}}}{2^n} + \frac{2q_p}{2^n} + \frac{3q_d}{2^n} + \frac{2q_d}{2^\tau}, \quad (7.10)$$

$$\text{Adv}_{\text{Remus-N2}}^{\text{auth}}(\mathcal{B}) \leq \frac{9\sigma_{\text{auth}}^2 + 4q_p\sigma_{\text{auth}}}{2^{2n}} + \frac{2q_p}{2^n} + \frac{3q_d}{2^n} + \frac{2q_d}{2^\tau}, \quad (7.11)$$

$$\text{Adv}_{\text{Remus-N3}}^{\text{auth}}(\mathcal{B}) \leq \frac{0.5\sigma_{\text{auth}}^2}{2^{k-8}} + \frac{q_p\sigma_{\text{auth}}}{2^k} + \frac{3q_d}{2^n} + \frac{2q_d}{2^\tau}. \quad (7.12)$$

Security of Remus-M

For the encryption query (N, A, M) or decryption query (N, A, C, T) of a AD blocks and m message blocks, the number of total TBC calls is at most $a + 2m$, which is called the number of *effective blocks* of a query.

Security Bound 7.5

Let $\mathcal{A}$ be an NR privacy adversary against Remus-M using q_e encryption (construction) queries and q_p primitive queries with total number of effective blocks in encryption queries σ_{priv} . Moreover, let $\mathcal{B}$ be an NR authenticity adversary using q_e encryption queries and q_d decryption queries, with total

7. Lightweight AEAD from Tweakable Block Ciphers

number of effective blocks for encryption and decryption queries σ_{auth} , and q_p primitive queries. Then we have

$$\text{Adv}_{\text{Remus-M1}}^{\text{priv}}(\mathcal{A}) \leq \frac{9\sigma_{\text{priv}}^2 + 4q_p\sigma_{\text{priv}}}{2^n} + \frac{2q_p}{2^n}, \quad (7.13)$$

$$\text{Adv}_{\text{Remus-M2}}^{\text{priv}}(\mathcal{A}) \leq \frac{9\sigma_{\text{priv}}^2 + 4q_p\sigma_{\text{priv}}}{2^{2n}} + \frac{2q_p}{2^n}, \quad (7.14)$$

$$\text{Adv}_{\text{Remus-M1}}^{\text{auth}}(\mathcal{B}) \leq \frac{9\sigma_{\text{auth}}^2 + 4q_p\sigma_{\text{auth}}}{2^n} + \frac{2q_p}{2^n} + \frac{5q_d}{2^n}, \quad (7.15)$$

$$\text{Adv}_{\text{Remus-M2}}^{\text{auth}}(\mathcal{B}) \leq \frac{9\sigma_{\text{auth}}^2 + 4q_p\sigma_{\text{auth}}}{2^{2n}} + \frac{2q_p}{2^n} + \frac{5q_d}{2^n}. \quad (7.16)$$

Security Bound 7.6

Let $\mathcal{A}$ be an NM privacy adversary and let $\mathcal{B}$ be an NM authenticity adversary, both are against Remus-M, that use the parameters as specified in Security Bound 5, and can repeat a nonce at most $1 \leq r \leq 2^{n-1}$ times in encryption queries. Then we have

$$\text{Adv}_{\text{Remus-M1}}^{\text{nm-priv}}(\mathcal{A}) \leq \frac{9\sigma_{\text{priv}}^2 + 4q_p\sigma_{\text{priv}}}{2^n} + \frac{2q_p}{2^n} + \frac{4r\sigma_{\text{priv}}}{2^n}, \quad (7.17)$$

$$\text{Adv}_{\text{Remus-M2}}^{\text{nm-priv}}(\mathcal{A}) \leq \frac{9\sigma_{\text{priv}}^2 + 4q_p\sigma_{\text{priv}}}{2^{2n}} + \frac{2q_p}{2^n} + \frac{4r\sigma_{\text{priv}}}{2^n}, \quad (7.18)$$

$$\text{Adv}_{\text{Remus-M1}}^{\text{nm-auth}}(\mathcal{B}) \leq \frac{9\sigma_{\text{auth}}^2 + 4q_p\sigma_{\text{auth}}}{2^n} + \frac{2q_p}{2^n} + \frac{4rq_e}{2^n} + \frac{5rq_d}{2^n}, \quad (7.19)$$

$$\text{Adv}_{\text{Remus-M2}}^{\text{nm-auth}}(\mathcal{B}) \leq \frac{9\sigma_{\text{auth}}^2 + 4q_p\sigma_{\text{auth}}}{2^{2n}} + \frac{2q_p}{2^n} + \frac{4rq_e}{2^n} + \frac{5rq_d}{2^n}. \quad (7.20)$$

We note that, as with the case of Romulus-M, Security Bounds 7.5 and 7.6 can be unified. We separate them for the difference of the security model and clarity of the proofs.

Bit Security. For NAE variants, Remus-N1 and Remus-N2 have $n/2$ -bit and n -bit security, while Remus-N3 has $\min\{(k-8)/2, n\}$ -bit security, for NR-privacy and NR-authenticity. For MRAE variants, Remus-M1 has $n/2$ -bit security for all relevant notions (NR/NM-privacy and authenticity), and Remus-M2 has n -bit security for NR-notions and $n/2$ -bit security for NM-notions with a graceful degradation with respect to nonce repetition in a similar manner to Romulus-M.

7.2.3. Remarks

All the security bounds hold as long as G is sound. As explained earlier, it is also straightforward to extend them to inputs of any bit strings rather than bytes (see Section 7.1.2). For all schemes, the basic proof strategy is similar to the proofs of iCOFB [46], and the recent proposal of Naito and Sugawara's PFB mode [109], but we needed a careful case analysis.

The Use of ICM. ICM has been acknowledged as a meaningful security proof model, especially in the field of hash function constructions. As described in the introduction, we can also obtain standard model proofs for Remus, by assuming the intermediate TBC ICE as a keyed primitive called tweakable pseudorandom permutation (TPRP). The standard model proofs are actually a part of ICM proofs. Specifically, the security bounds under this standard model will appear in the hybrid argument of ICM proofs. A similar technique appeared in some permutation-based schemes [130, 131, 132], where the keyed primitive is a variant of the Even-Mansour cipher. We warn that the assumption that ICE is a TPRP does not imply its perfect security: there are generic attacks which work even if the underlying TBC is an ideal cipher. Therefore, the expected bit security levels are identical for both ICM and this standard model analyses, and we only present the security bounds under ICM.

Tightness of the Security Bounds. We note that our security bounds are tight. Indeed, the existing third-party analysis for Remus-N3 and for Remus-N1 and Remus-M1 [133] show their tightness. We emphasize that these analysis do not contradict our security bounds. The attacks exploit the fact that the initial values L and V are randomised, and can be targeted by an offline-online matching attack. For example, an adversary can perform the following attack:

1. The adversary encrypts on his own machine a message M with a set of different values of (L, V) .
2. The adversary asks for the encryption of M with a set different nonces N .
3. The adversary tries to find a colliding ciphertext between his own encryptions and the encryption obtained from the oracle.
4. If the collision is detected, the adversary knows the values of (L, V) with high probability and can use to to forge a new message.

If the size of the first set is t and the size of the second set is q , then the success probability of the attack is high when $q \times t \approx 2^{|L|+|V|}$. In other words, for Remus-N1, $q \times t \approx 2^n$, while for Remus-N2, $q \times t \approx 2^{2n}$. In case of Remus-N3, the value L is linearly related to the master key. hence, the attack can be extended to master key recovery. A user of Remus must choose the appropriate parameters for their use case.

7.3. Design Rationale of Romulus and Remus

Rationale of the NAE Modes. Romulus-N has a similar structure as a mode called iCOFB, which appeared in the full version of CHES 2017 paper [46]. Because it was introduced to show the feasibility of the main proposal of [46], block cipher mode COFB, it does not work as a full-fledged AE using conventional TBCs. Therefore, starting from iCOFB, we apply numerous changes for improving efficiency while achieving high security. As a result, Romulus-N becomes a much more advanced, sophisticated NAE mode based on a TBC. The security bound of Romulus-N is essentially equivalent to Θ CB3, having full n -bit security, since the bounds only differ in their constants.

7. Lightweight AEAD from Tweakable Block Ciphers

Recently, a new mode called PFB was published by Naito and Sugawara [109], which holds some similarities to Romulus-N and shares some of its design goals. It uses different feedback functions for the associated data, plaintext and ciphertext, with the purpose of providing partial parallelizability for the encryption algorithm, where the associated data processing and the ciphertext decryption is serial in nature, while the plaintext encryption can be parallelized. Since our goal in Romulus and Remus is to achieve very lightweight hardware implementations, we chose a fixed feedback function for all the parts of the algorithm, in order to avoid additional multiplexers, given that applications of such partial parallelizability are limited. Moreover, it was shown during the CAESAR competition that the parallelized hardware implementations of block-cipher based modes, e.g. Deoxys, does not provide huge performance gains as in the case of software implementations [75].

By seeing Remus-N as a mode of a TBC (ICE), Remus-N also has a similar structure as iCOFB [46]. Assuming ICE is an ideally secure TBC, the security bound of Remus-N is also essentially equivalent to ΘCB3 , having full n -bit security. The remaining problem is how to efficiently instantiate a TBC. We could use a dedicated TBC, or a conventional block cipher mode (such as XEX [41]), however, they have certain limitations on security and efficiency. To overcome such limitations, we choose to use a block cipher with tweak-dependent key/mask derivation. This approach, initiated by Mennink [129], improves the performance over earlier approaches, at the expense of the ideal cipher model for security. Specifically, the TBC ICE has three variants, where ICE1 and ICE2 can be seen as variants of XHX [134], and ICE3 is a more classical one combined with a doubling mask [41]. Each variant has its own security level, namely, ICE1 has $n/2$ -bit security, ICE2 has n -bit security, and ICE3 has $(n/2 - 4)$ -bit security. They have different computation cost for key/mask derivations and have different state sizes. Given the n -bit security of *outer* TBC-based mode, the standard hybrid argument shows that the security of Remus-N is effectively determined by the security of the internal ICE.

Rationale of the MRAE Modes. Romulus-M and Remus-M are designed as MRAE modes following the structure of SIV [44] and SCT [11]. Romulus-M reuses the components of Romulus-N and Remus-M reuses the components of Remus-N as much as possible to inherit their implementation advantages and the security. In fact, this brings us several advantages (not only for implementation aspects) over SIV/SCT. Compared with SCT, Romulus-M needs fewer primitive calls because of the faster MAC. Moreover, Romulus-M has a smaller state than SCT because of the single-state encryption part taken from Romulus-N (SCT employs a variant of counter mode). Remus-M needs an equivalent number of primitive calls as SCT. The difference is in the primitive: Remus-M uses an n -bit block cipher while SCT uses an n -bit-block dedicated TBC. Moreover, Remus-M has a smaller state than SCT because of the single-state encryption part taken from Remus-N.

The provable security of Romulus-M is similar to SCT: the security depends on the maximum number of repetition of a nonce in encryption (r), and if $r = 1$ (i.e., NR adversary) we have the full n -bit security. Security will gradually decrease as r increases, also known as “graceful degradation”, and even if r equals the number of encryption queries, implying nonces are fixed, we maintain the birthday-bound, $n/2$ -bit security. Similarly to Remus-N, the provable security of Remus-M is effectively determined by the internal ICE. For Remus-M2, thanks to n -bit security of ICE2, its security is also similar to SCT. For NR-adversaries, we have

the full n -bit security. Security will gradually decrease as r increases, and even if r equals the number of encryption queries, we maintain the birthday-bound, $n/2$ -bit security. For Remus-M1, the security is $n/2$ bits for both NR and NM adversaries due to the $n/2$ -bit security of ICE1.

ZAE [113] is another TBC-based MRAE. Although it is faster than SCT, the state size is much larger than SCT, Romulus-M and Remus-M.

Theoretical Efficiency Comparison. In Table 5.1, we compare a group of recently proposed lightweight AEAD modes. Our comparative analysis shows that Romulus-N is smaller and more efficient than ΘCB3 for the same security level. Moreover, the cost of processing AD is about half that of the message. For example, in the case of Romulus-N1, if the message and AD have equal length, there is an extra speed up of $\sim 1.33x$, which means that the efficiency even increases from 3.5λ to 2.625λ , compared to 4.5λ in case of ΘCB3 , which makes Romulus-N a very promising candidate for NAE, for both short and long messages. Besides, it shows that Remus-N achieves its goals, as Remus-N1 has a $2n$ -bit state, which is smaller than COFB and equal to Beetle. Remus-N1 and COFB both have birthday security, i.e., $n/2$. Beetle achieves higher security, at the expense of using a $2n$ -bit permutation. Our analysis also shows that among the considered AEAD modes, Remus-N2 achieves the lowest R/S ratio, with a state size of $3n$ but only an n -bit permutation. Since Remus-N3 uses a 64-bit block cipher, we manage to achieve very small area and more relaxed state size.

7.4. Hardware Implementations

7.4.1. General Architecture and Hardware Estimates

The goal of the design of Romulus and Remus is to have a very small area overhead over the underlying TBC, specially for the round-based implementations. In order to achieve this goal, we set two requirements:

1. There should be no extra Flip-Flops over what is already required by the TBC, since Flip-Flops are very costly ($4 \sim 7$ GEs per Flip-Flop).
2. The number of possible inputs to each Flip-Flop and outputs of the circuits have to be minimized. This is in order to reduce the number of multiplexers required, which is usually one of the causes of efficiency reduction between the specification and implementation.

One of the advantages of Skinny as a lightweight TBC is that it has a very simple datapath, consisting of a simple state register followed by a low-area combinational circuit, where the same circuit is used for all the rounds, so the only multiplexer required is to select between the initial input for the first round and the round output afterwards (Figure 4.1(a)), and it has been shown that this multiplexer can even have lower cost than a normal multiplexer if it is combined with the Flip-Flops by using Scan-Flops (Figure 4.1(b)) [135, 22]. However, when used inside an AEAD mode, challenges arise, such as how to store the key and nonce, as the key scheduling algorithm will change these values after each block encryption. The same goes for the block counter. In order to avoid duplicating the storage elements for these values; one set to be used to execute the TBC and one set to be

7. Lightweight AEAD from Tweakable Block Ciphers

used by the mode to maintain the current value, we studied the relation between the original and final value of the tweakkey. Since the key scheduling algorithm of Skinny is fully linear and has very low area (most of the algorithm is just routing and renaming of different bytes), the full algorithm can be inverted using a very small circuit that costs 64 XOR gates for **Romulus-N1** and 0 gates for **Remus**. Moreover, the LFSR computation required between blocks can be implemented on top of this circuit, costing 3 extra XOR gates. This operation can be computed in parallel to ρ , such that when the state is updated for the next block, the tweakkey key required is also ready. This costs only ~ 67 XOR gates as opposed to ~ 320 Flip-Flops that will, otherwise, be needed to maintain the tweakkey value. Hence, the mode was designed with the architecture in Figure 4.1(b) in mind, where only a full-width state-register is used, carrying the TBC state and tweakkey values, and every cycle, it is either kept without change, updated with the TBC round output (which includes a single round of the key scheduling algorithm) or the output of a simple linear transformation, which consists of ρ/ρ^{-1} , the unrolled inverse key schedule and the block counter. In order estimate the hardware cost of **Romulus-N1** the mode we consider the round based implementation with an $n/4$ -bit input/output bus:

- 4 XOR gates for computing G .
- 64 XOR gates for computing ρ .
- 67 XOR gates for the correction of the tweakkey and counting.
- 56 multiplexers to select whether to choose to increment the counter or not.
- 320 multiplexers to select between the output of the Skinny round and lt .

This adds up to 135 XOR gates and 376 multiplexers. For estimation purposes assume an XOR gate costs 2.25 GEs and a multiplexer costs 2.75 GEs, which adds up to 1337.75 GEs. In the original Skinny paper [18], the authors reported that Skinny-128-384 requires 4,268 GEs, which adds up to $\sim 5,605$ GEs. This is ~ 1.4 KGEs smaller than the round based implementation of Ascon [136]. Moreover, a smart design can make use of the fact that 64 bits of the tweakkey of Skinny-128-384 are not used, replacing 64 Flip-Flops by 64 multiplexers reducing an extra ~ 200 GEs. In order to design a combined encryption/decryption circuit, we show below that the decryption costs only extra 32 multiplexers and ~ 32 OR gates, or ~ 100 GEs. A similar analysis is done for **Romulus-N2** and **Romulus-N3**, estimating that they would cost 1,217 and 1,073 GEs, respectively, on top of the corresponding Skinny variants, or 5,485 and 4,385 GEs, respectively.

These estimations show that **Romulus-N** is not just competitive theoretically but it can be a very attractive option practically for low-area applications. For example, the 8-bit implementation of ACORN, the smallest implementation publicly available for all the round 3 candidates of the CAESAR competition, costs 5,900 GEs, as shown in [137]. If we assume around $\sim 1,000$ GEs as the cost of the CAESAR Hardware API included in that design, as reported in [136], then **Romulus-N3** is still smaller than that. Besides, the area can be even lower using serial implementations of Skinny, which cost $\sim 3,000$ GE for Skinny-128-384 and $\sim 2,000$ GEs for Skinny-128-256, a gain of more than 1,000 GEs compared to the round-based implementation.

Figure 7.13.: Serial state update function used in Romulus and Remus.

Another possible optimization is to consider the fact that most of the area of Skinny comes from the storage elements, hence, we can speed up Romulus to almost double the speed by using a simple two-round unrolling, which costs $\sim 1,000$ GEs, as only the logic part of Skinny needs replication, which is only $< 20\%$ increase in terms of area.

Romulus-M is estimated to have almost the same area as Romulus-N, except for an additional set of multiplexers in order to use the tag as an initial vector for the encryption part. This indicates that it can be a very lightweight choice for high security applications.

For the serial implementations we followed the currently popular bit-sliding framework [22] with minor tweaks. The state of Skinny is represented as a Feedback-Shift Register which typically operates on 8 bits at a time, while allowing the 32-bit MixColumns operation, given in Figure 7.13.

It can be viewed in Figure 7.13 that several careful design choices such as a lightweight serializable ρ function without the need of any extra storage and a lightweight padding/truncation scheme allow the low-area implementations to use a very small number of multiplexers on top of the Skinny circuit for the state update, three 8-bit multiplexers to be exact, two of which have a constant zero input, and ~ 22 XORs for the ρ function and block counter. For the key update functions, we did several experiments on how to serialize the operations and we found that the best trade-off is to design a parallel/serial register for every tweakkey, where the key schedule and mode operations are done in the same manner of the round based implementation, while the AddRoundTweakey operation of Skinny is done serial as shown in Figure 7.13.

Usually there is a disparity between the theoretical estimate of the state size of a mode and the practical implementations. However, our practical implementations show that our theoretical estimates match exactly our implementations for both round-based and serial implementations up to a constant term that covers the round constants of Skinny and the Finite State Machine of the API/interface, as shown in Table 7.1.

7. Lightweight AEAD from Tweakable Block Ciphers

Table 7.1.: Practical state size of Romulus and Remus variants.

Variant	State Size
Romulus-N1	$448 + \mathcal{O}(1)$
Remus-N1	$256 + \mathcal{O}(1)$
Remus-N2	$384 + \mathcal{O}(1)$
Remus-M1	$256 + \mathcal{O}(1)$
Remus-M2	$384 + \mathcal{O}(1)$

7.4.2. Hardware Implementations Costs

Hardware Cost of Remus-N1. The overhead of Remus-N1 is mostly due to the doubling (3 XORs) and ρ operations (68 XORs, assuming the input/output bus has width of 32 bits). Moreover, we need 2 128-bit multiplexers to select the input to the tweakkey out of four positive values: K , S (after applying the KDF function), lt , or the Skinny round key. We assume a multiplexer costs ~ 2.75 GEs and an XOR gate costs ~ 2.25 GEs. In total, this adds up to ~ 864 GEs on top of Skinny-128/128. In the original Skinny paper [18], the authors reported that the round-based implementation of Skinny-128/128 costs $\sim 2,391$ GEs. So we estimate that Remus-N1 should cost $\sim 3,255$ GEs, which is a very small figure, compared to not just TBC based AEAD modes, but in general. For example, ACORN, the smallest CAESAR winner, costs $\sim 5,900$ GEs. Besides, two further optimizations are applicable. First, we can use the serial Skinny-128/128 implementation, which costs ~ 600 GEs less. The other direction is to unroll Skinny-128/128 to a 2- or 4-round implementation, reducing the number of cycles, at the cost of 1 KGEs per extra round. We do acknowledge that this huge gain in area comes at the cost of reduced security (birthday security). In order to design a combined encryption/decryption circuit, we show below that the decryption costs only extra 32 multiplexers and ~ 32 OR gates, or ~ 100 GEs.

Hardware Cost of Remus-N2. Remus-N2 is similar to Remus-N1, with an additional mask V . Hence, the additional cost comes from the need to store and process this mask. The storage cost is simply 128 extra Flip-Flops. However, the processing cost can be tricky, especially since we adopt a serial concept for the implementation of ρ . Hence, we also adopt a serial concept for the processing of V . We assume that V will be updated in parallel to ρ and we merge the masking and ρ operations. Consequently, we need 64 XORs for the masking, 3 XORs for doubling, 5 XORs in order to correct the domain separation bits after each block (note that 3 bits are fixed), and 1 Flip-Flop for the serialization of doubling. Overall, we need ~ 800 GEs on top of Remus-N1, $\sim 4,055$ GEs overall, which is again smaller than almost all other AEAD designs (except other Remus variants), while achieving BBB security.

Hardware Cost for Remus-N3. Remus-N3 uses Skinny-64/128 as the TBC (via the mode ICE3). According to our estimations, it requires 224 multiplexers, 32 XOR gates for the KDF function, 68 XORs for ρ , 24 XOR gates for correcting the Tweakkey and 3 XOR gates for the counting. This adds up to ~ 743 GEs on top of Skinny-64/128, which costs 1,399 GEs. So we estimate that Remus-N3 costs

2,439 GEs, which is a very small figure for any round-based implementation of an AEAD mode.

The arguments about serialization, unrolling and decryption are the same for all Remus-N variants. Thanks to the shared structure, these arguments also generally apply to Remus-M.

7.4.3. Primitives Choices

LFSR-Based Counters. The NIST call for lightweight AEAD algorithms requires that such algorithms must allow encrypting messages of length at least 2^{50} bytes while still maintaining their security claims. This means that using a TBC whose block size is 128 bits, we need a block counter of a period of at least 2^{46} . While this can be achieved by a simple arithmetic counter of 46 bits, arithmetic counters can be costly both in terms of area ($3 \sim 5$ GEs/bit) and performance (due to the long carry chains which limit the frequency of the circuit). In order to avoid this, we decided to use LFSR-based counters which can be implemented using a handful of XOR gates ($3 \text{ XORs} \approx 6 \sim 9$ GEs). This, in addition to the architecture described above, makes the cost of counter almost negligible. For Remus, these counters are either dedicated counters, in the case of Romulus, or consecutive doubling of the key L which is equivalent to a Galois LFSR, in the case of Remus. This, in addition to the architecture described above, makes the cost of counters almost negligible.

Counter Separation (Romulus-N2 and Romulus-M2). For Romulus-N2 and Romulus-M2, we used two LFSRs instead of one, such that the second LFSR is updated only when the first LFSR performs a full period. The rationale for that is to provide an even more lightweight variant of Romulus-N1 and Romulus-M1. The maximum input length imposed by NIST is quite large. In practice, the users may only need a support of input length up to $\sim 2^{28}$ bytes instead. In this case, the second LFSR does not need to be implemented. This saves $64 \sim 128$ Flip-Flops.

Smaller $\mathcal{D}$ for Romulus-N3. The goal of Romulus-N3 is to fit the Remus algorithm in Skinny-128-256, which is faster and smaller than Skinny-128-384. In most lightweight applications, the amount of data to be sent under the same key is small. Hence, Romulus-N3 represents a variant targeted at such applications that is faster and smaller than the other variants and can encrypt up to ~ 256 MBs of data.

Tag Generation. Considering hardware simplicity, the tag is the final output state (i.e., the same way as the ciphertext blocks), as opposed to the final state S of the TBC. In order to avoid branching when it comes to the output of the circuit, the tag is generated as $G(S)$ instead of S . In hardware, this can be implemented as $\rho(S, 0^n)$, i.e., similar to the encryption of a zero vector. Consequently, the output bus is always connected to the output of ρ and a multiplexer is avoided.

Padding. The padding function used in Romulus and Remus is described as follows:

7. Lightweight AEAD from Tweakable Block Ciphers

For $X \in \{0, 1\}^{\leq l}$ of length multiple of 8 (i.e, byte string), let

$$\text{pad}_l(X) = \begin{cases} X & \text{if } |X| = l, \\ X \parallel 0^{l-|X|-8} \parallel \text{len}_8(X) & \text{if } 0 \leq |X| < l, \end{cases}$$

where $\text{len}_8(X)$ denotes the one-byte encoding of the byte-length of X (we assume that $l < 256$ bytes). Here, $\text{pad}_l(\varepsilon) = 0^l$. For example, when $l = 128$, $\text{len}_8(X)$ has 16 variations (i.e., byte length 0 to 15), and we encode it to the last 4 bits of $\text{len}_8(X)$ (for example, $\text{len}_8(11) = 00001011$). The case $l = 64$ is similarly treated, by using the last 3 bits.

The padding function used in Romulus and Remus is chosen so that the padding information is always inserted in the most significant byte of the last block of the message/AD. Hence, it reduces the number of decisions for each byte to only two decisions (either the input byte or a zero byte, except the most significant byte which is either the input byte or the byte length of that block). Besides, it is also the case when the input is treated as a string of words (16-, 32-, 64- or 128-bit words). This is much simpler than the classical 10^* padding approach, where every word has a lot of different possibilities when it comes to the location of the padding string. Besides, usually implementations maintain the length of the message in a local variable/register, which means that the padding information is already available, just a matter of placing it in the right place in the message, as opposed to the decoder required to convert the message length into 10^* padding.

Padding Circuit for Decryption. One of the main features of Remus is that it is inverse free and both the encryption and decryption algorithms are almost the same. However, it can be tricky to understand the behavior of decryption when the last ciphertext block has length $< n$. In order to understand padding in the decryption algorithm, we look at the ρ and ρ^{-1} functions when the input plaintext/ciphertext is partial. The ρ function applied on a partial plaintext block is shown in Equation (7.21). If ρ^{-1} is directly applied to $\text{pad}_n(C)$, the corresponding output will be incorrect, due to the truncation of the last ciphertext block. Hence, before applying ρ^{-1} we need to regenerate the truncated bits. It can be verified that $C' = \text{pad}_n(C) \oplus \text{msb}_{n-|C|}(G(S))$. Once C' is regenerated, ρ^{-1} can be computed as shown in Equation (7.22):

$$\begin{bmatrix} S' \\ C' \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ G & 1 \end{bmatrix} \begin{bmatrix} S \\ \text{pad}_n(M) \end{bmatrix} \quad \text{and} \quad C = \text{lsb}_{|M|}(C'). \quad (7.21)$$

$$C' = \text{pad}_n(C) \oplus \text{msb}_{n-|C|}(G(S)) \quad \text{and} \quad \begin{bmatrix} S' \\ M \end{bmatrix} = \begin{bmatrix} 1 \oplus G & 1 \\ G & 1 \end{bmatrix} \begin{bmatrix} S \\ C' \end{bmatrix}. \quad (7.22)$$

While this looks like a special padding function, in practice it is simple. First of all, $G(S)$ needs to be calculated anyway. Besides, the whole operation can be implemented in two steps:

$$M = C \oplus \text{lsb}_{|C|}(G(s)), \quad (7.23)$$

$$S' = \text{pad}_n(M) \oplus S \quad (7.24)$$

which can have a very simple hardware implementation, as discussed in the next paragraph.

Encryption-Decryption Combined Circuit. One of the goals of Romulus and Remus is to be efficient for implementations that require a combined encryption-decryption datapath. Hence, we made sure that the algorithm is inverse free, i.e., it does not use the inverse function of Skinny or $G(S)$. Moreover, ρ and ρ^{-1} can be implemented and combined using only one multiplexer, whose size depends on the size of the input/output bus. The same circuit can be used to solve the padding issue in decryption, by padding M instead of C . The tag verification operation simply checks if $\rho(S, 0^n)$ equals to T , which can be serialized depending on the implementation of ρ .

Choice of the G Matrix. We chose the position of G so that it is applied to the output state. This removes the need of G for AD processing, which improves software performance. In Section 7.1.2, we listed the security condition for G , and we choose our matrix G so that it meets these conditions and suits well for various hardware and software platforms.

We noticed that for lightweight applications, most implementations use an input/output bus of width ≤ 32 . Hence, we expect the implementation of ρ to be serialized depending on the bus size. Consequently, the matrix used in iCOFB can be inefficient as it needs a feedback operation over 4 bytes, which requires up to 32 extra Flip-Flops in order to be serialized, something we are trying to avoid in Remus. Moreover, the serial operation of ρ is different for byte, which requires additional multiplexers.

However, we observed that if the input block is interpreted in a different order, both problems can be avoided. First, it is impossible to satisfy the security requirements of G without any feedback signals, i.e., G is a bit permutation.

- If G is a bit permutation with at least one bit going to itself, then there is at least one non-zero value on the diagonal, so $I + G$ has at least 1 row that is all 0s.
- If G is a bit permutation without any bit going to itself, then every column in $I + G$ has exactly two 1's. The sum of all rows in such matrix is the 0 vector, which means the rows are linearly dependent. Hence, $I + G$ is not invertible.

However, the number of feedback signals can be adjusted to our requirements, starting from only 1 feedback signal. Second, we noticed that the input block/state of length n bits can be treated as several independent sub-blocks of size n/w each. Hence, it is enough to design a matrix G_s of size $w \times w$ bits and apply it independently n/w times to each sub-block. The operation applied on each sub-block in this case is the same (i.e., as we can distribute the feedback bits evenly across the input block). Unfortunately, the choice of w and G_s that provides the optimal results depends on the implementation architecture. However, we found out that the best trade-off/balance across different architectures is when $w = 8$ and G_s uses a single bit feedback.

In order to verify our observations, we generated a family of matrices with different values of w and G_s , and measured the cost of implementing each of them on different architectures.

7.5. Instantiation and Efficiency of Romulus and Remus

7.5.1. Instantiations of Romulus and Remus

We instantiated our two modes Romulus and Remus with the lightweight tweakable block cipher Skinny [18], which has the very interesting feature that if some part of the tweak is not used, the corresponding area for storage can be avoided. We tried to minimize the global control logic and the multiplexers. More details about the instantiations can be found in Section 7.7.

7.5.2. Software efficiency

We refer to the Skinny document for discussions on software implementations of the various Skinny versions [18]. The Romulus mode will have little impact on the global performance of Skinny in software as long as serial implementations are used. We expect very little increase in ROM or RAM when compared to Skinny benchmarks. The very performant micro-controller implementations reported in the Skinny document were benchmarked without assuming parallel cipher calls, and without any pre-processing. Therefore, Romulus will present a very similar performance profile as the numbers reported on micro-controllers. Generally, using little amount of RAM, Skinny is easy and efficient to implement using a simple table-based approach.

For high-end platforms, such as latest Intel processors, very efficient highly-parallel bitsliced implementations of Skinny using SSE, AVX, AVX2 instructions on XMM/YMM registers will not be directly applicable as our Romulus mode is serial in nature. However, in a classical case of a server communicating with many lightweight devices, we note that it would be possible to consider bitslicing the key schedule [138] of Skinny (being relatively simple to compute) or using scheduling strategies [139]. Classical table-based implementations of Skinny will ensure acceptable performance on even legacy platforms, while Vector Permute (vperm) might lead to better results on medium range platforms by parallelizing the computation of the Sbox.

7.5.3. ASIC Efficiency

We have implemented several variants of the Romulus and Remus families in the TSMC 65nm standard cell library, in order to have a better understanding of the ASIC performance and cost and verify our estimations in Section 7.4. The implementations use the round-based Skinny implementation published on the Skinny website [140]. The results in Table 4.6 show that Remus is very lightweight and efficient at the same time as it requires slightly above 3 KGE (3.5 KGE with a simple interface) for its round-based implementation for Remus-N1. Of course, even smaller (but slower) trade-offs are possible by going with a serial implementation. At the other end of the spectrum in terms of security, the nonce misuse-resistant version Remus-M2 can be implemented in less than 5 KGE and provides 128-bit security, even in environments where randomness is not very reliable.

We have also implemented the serial, round-based and 4-round unrolled architectures of Romulus-N1. The figures are expected to be similar for Romulus-N2 and about 550 GEs smaller, and 12~18% faster for Romulus-N3. Moreover, Remus and Romulus share a similar structure and our experimental results show that

it is very cheap to convert the implementation to the misuse-resistant variant. The implementation results of the Romulus-N1 and Remus-N1 4-round unrolled architecture show how Romulus and Remus take advantage of the versatility of Skinny for different architectures, where the throughput can be multiplied by 4 and the efficiency can be increased at a smaller area cost (about 75~100%).

We have implemented Romulus-N1, Remus-N1 and Remus-N2 using the byte serial Skinny architecture and serialized feedback function for the modes. Serial implementations for different variants of Skinny have been proposed in [18, 140, 22]. Such implementations can be easily adapted for other Romulus and Remus members. We have implemented one of our smallest proposals, Remus-N1 using the byte-serial implementation of Skinny. Without the interface, Remus-N1 requires around 2100 GEs using the TSMC 65nm standard cell library including the Skinny logic, Remus-N1 Logic and both the key and state storage.

We compare our implementations to the implementations of Ascon and ACORN, the winners of the lightweight portfolio of the CAESAR competition. We have downloaded the implementation of ACORN from the ATHENa benchmarking website [80] that were also used as part of the ASIC benchmarking efforts of the CAESAR competition [137]. For Ascon, we have downloaded the serial and round based implementations from the designers' website [141], since these implementations seem to have better performance metrics compared to the one on the ATHENa website, at the expense of using a different interface. We have synthesized all four implementations to the same technology we are using, TSMC 65nm, to allow fair comparisons. We show that Romulus achieves comparable performance (but Romulus security proofs are in the standard model) while Remus is very competitive in terms of low area. For example, all the variants of Remus can achieve throughput above 1 Gbps for less than 5000 GEs, while the performance of Ascon drops significantly when the area is reduced to this range. Additionally, we believe the different security models and goals should be taken into consideration when comparing the performance, as in Romulus and Remus we use a much stronger primitive without any round reduction. However, we must note that Ascon is significantly better for energy efficiency using the round-based architecture because of this difference, as the round based implementation needs only 6 cycles per primitive call. A more thorough exploration of the design space of Romulus-N1 is provided in Section 7.6.

7.5.4. FPGA Efficiency

The FPGA results presented in Tables 7.2 show that the FPGA implementations of Romulus and Remus follow the same trend as the ASIC implementations achieving between 156 and 285 Slices for the round based implementations and less than 500 Slices for the 4-round unrolled implementations.

Table 7.2.: FPGA Results for Remus and Romulus on the Xilinx Virtex 6 FPGA using ISE

Variant	Slices	LUTs	Registers	Max. Freq. (MHz)	Throughput (Mbps)	Throughput/Area (Mbps/Area)
Nonce-respecting schemes (64-bit data security)						
Remus-N1	189	540	308	250	727.7	3.8
Remus-N1 [†]	550	1669	338	147.7	1358.84	2.4
Nonce-respecting schemes (128-bit data security)						
Remus-N2	225	757	358	250	727.7	3.23
Romulus-N1	307	919	534	250	695	2.26
Romulus-N1 [†]	597	1884	528	250	2300	3.85
Lilliput-I-128	391	1506	1017	185	657.8	1.68
Lilliput-II-128	309	1088	885	185	328.9	1.06
Nonce-misuse resistant schemes (64-bit data security)						
Remus-M1	220	595	322	240	348	1.58
Nonce-misuse resistant schemes (128-bit data security)						
Remus-M2	279	816	397	219	317.6	1.13

[†] Unrolled x4.

7.6. ASIC Design Space Exploration of Romulus-N1

In this section we provide a full list of different ASIC implementation trade-offs Romulus-N1, showing the design range of our proposals. First we study the impact of different number of round-unrolling, where Rx means round-based implementation with x round unrolling. We vary the operating frequency from the maximum possible frequency till 125 MHz and see the impact on throughput, area, power and energy. The results are given in Table 7.3. Second, in Table 7.4, we study the byte serial architecture based on the bit-sliding technique, following the same methodology. Finally, in Table 7.5, we study the 3-share threshold implementation of both the byte serial architecture (PS) and the single round architecture (P1). The implementations are based on the threshold implementations provided by the Skinny team [18]. The results show that the best throughput/area trade off is achieved by the R2 architecture, which processes two rounds on the TBC per cycle. The minimum energy is achieved by 4-round unrolling (the R4 architecture), as it requires only 18 cycles per block for encryption and 11 cycles per block for associated data. The serial implementation can be as low as 3.3 KGE, which is extremely low for a mode with full n -bit security and standard model security proofs. On the other hand, for about 8 KGE, we can have a performant threshold implementation protected against side-channel attacks.

Table 7.3.: The design space of Romulus-N1 using the TSMC-65 ASIC technology: (unrolled) round-based implementations

Arch.	Critical Path (ns)	Area (GE)	Power (mW)	Energy (Enc block) (pJ)	Energy (Auth block) (pJ)	Throughput (Enc only) (Mbps)	Throughput (Auth only) (Mbps)	Throughput ($ A = M $) (Mbps)	Thput/Area
R1	0.5	6220	0.8	24	12.8	4267	8000	6133	0.98
R1	0.66	5877	0.72	28.5	15.2	3232	6060	4647	0.79
R1	0.75	5864	0.65	29.25	15.6	2844	5333	4089	0.69
R1	1	5860	0.53	31.8	16.9	2133	4000	3067	0.52
R1	2	5860	0.35	42	22.4	1067	2000	1533	0.26
R1	4	5860	0.25	60	32	533	1000	767	0.13
R1	8	5772	0.22	105.6	56.3	266	500	383	0.07
R2	0.5	7978	0.99	15.84	8.91	8000	14200	11100	1.39
R2	0.66	7161	0.78	16.47	9.2	6060	10758	8410	1.17
R2	0.75	6860	0.71	17.04	9.5	5333	9467	7400	1.07
R2	1	6727	0.56	17.92	10.08	4000	7100	5550	0.82
R2	2	6635	0.38	24.32	13.68	2000	3550	2775	0.41
R2	4	6635	0.29	60	20.88	1000	1775	1388	0.2
R2	8	6635	0.25	64	36	500	888	694	0.1
R4	0.8	12766	1.1	15.84	9.68	8888	14500	11694	0.91
R4	1	10008	0.77	13.86	8.47	7111	11600	9356	0.93
R4	2	8740	0.46	16.56	10.12	3555	5800	4678	0.53
R4	4	8740	0.36	25.92	15.84	1775	2900	2339	0.26
R4	8	8740	0.32	46.08	28.16	889	1450	1170	0.13
R8	1.6	18679	1.2	21.12	14.4	7250	10662	8956	0.48
R8	2	15133	0.89	19.58	13.35	5800	8530	7165	0.47
R8	4	12990	0.51	22.44	15.3	2900	4265	3582	0.28
R8	8	12990	0.45	39.6	27	1450	2132	1791	0.14

Table 7.4.: The design space of Romulus-N1 using the TSMC-65 ASIC technology: byte serial implementations

Arch.	Critical Path (ns)	Area (GE)	Power (mW)	Energy (Enc block) (pJ)	Energy (Auth block) (pJ)	Throughput (Enc only) (Mbps)	Throughput (Auth only) (Mbps)	Throughput ($ A = M $) (Mbps)	Thput/Area
S1	0.75	3390	0.5	489	247.5	131	259	195	0.06
S1	1	3318	0.5	652	330	98	194	146	0.04
S1	2	3318	0.29	756	383	49	97	73	0.02
S1	4	3318	0.2	1043	528	25	49	37	0.01
S1	8	3318	0.15	1565	792	12	24	18	0.005

Table 7.5.: The design space of Romulus-N1 using the TSMC-65 ASIC technology: byte serial implementations and round based first-order threshold implementations

Arch.	Critical Path (ns)	Area (GE)	Power (mW)	Energy (Enc block) (pJ)	Energy (Auth block) (pJ)	Throughput (Enc only) (Mbps)	Throughput (Auth only) (Mbps)	Throughput ($ A = M $) (Mbps)	Thput/Area
PS	0.75	5163	0.79	772	391	131	259	195	0.04
PS	1	5158	0.62	808	409	98	194	146	0.03
PS	2	5154	0.40	1043	529	49	97	73	0.01
PS	4	5154	0.27	1408	713	25	49	37	0.007
PS	8	5154	0.21	2190	1110	12	24	18	0.003
P1	0.5	8386	1.2	36	19.2	4267	8000	6133	0.73
P1	0.66	8101	0.89	35.2	18.7	3232	6060	4647	0.57
P1	0.75	8048	0.8	36	19.2	2844	5333	4089	0.51
P1	1	8048	0.69	41.4	22.1	2133	4000	3067	0.38
P1	2	8048	0.46	55.2	29.4	1067	2000	1533	0.19
P1	4	8048	0.35	84	44.8	533	1000	767	0.095
P1	8	8048	0.28	134.4	51.6	266	500	383	0.05

7.7. Instantiation of Romulus and Remus with Skinny

Endian. For both Romulus and Remus, we employ little endian for byte ordering: an n -bit string X is received as

$$X_7X_6 \dots X_0 \parallel X_{15}X_{14} \dots X_8 \parallel \dots \parallel X_{n-1}X_{n-2} \dots X_{n-8},$$

where X_i denotes the $(i + 1)$ -st bit of X (for $i \in \llbracket n \rrbracket_0$). Therefore, when c is a multiple of 8 and X is a byte string, $\text{msb}_c(X)$ and $\text{lsb}_c(X)$ denote the last (rightmost) c bytes of X and the first (leftmost) c bytes of X , respectively. For example, $\text{lsb}_{16}(X) = (X_7X_6 \dots X_0 \parallel X_{15}X_{14} \dots X_8)$ and $\text{msb}_8(X) = (X_{n-1}X_{n-2} \dots X_{n-8})$ with the above X . Since our specification is defined over byte strings, we only consider the above case for msb and lsb functions (i.e., the subscript c is always a multiple of 8). One can interpret lsb_i and msb_i as lmt_i and rmt_i in the specification of Section [7.1](#).

7.7.1. Instantiating Romulus with Skinny

We propose three versions of Romulus-N and three versions of Romulus-M when instantiated with Skinny. We provide in Table [7.6](#) the parameters for these variants.

The LFSR

We use LFSRs for counter. For positive integer c , lfsr_c is a one-to-one mapping $\text{lfsr}_c : \llbracket 2^c - 1 \rrbracket_0 \rightarrow \{0, 1\}^c \setminus \{0^c\}$ defined as follows. For positive integer c , let $F_c(\mathbf{x})$ be the lexicographically-first polynomial among the the irreducible degree c polynomials of a minimum number of coefficients. Specifically $F_c(\mathbf{x})$ for $c \in \{56, 24\}$ are

$$\begin{aligned} F_{56}(\mathbf{x}) &= \mathbf{x}^{56} + \mathbf{x}^7 + \mathbf{x}^4 + \mathbf{x}^2 + 1, \\ F_{24}(\mathbf{x}) &= \mathbf{x}^{24} + \mathbf{x}^4 + \mathbf{x}^3 + \mathbf{x} + 1, \end{aligned}$$

and

$$\text{lfsr}_c(D) = 2^D \bmod F_c(\mathbf{x}).$$

Note that we use $\text{lfsr}_c(D)$ as a block counter, so most of the time D changes incrementally with a step of 1, and this enables $\text{lfsr}_c(D)$ to generate a sequence of $2^c - 1$ pairwise-distinct values. From an implementation point of view, it should be implemented in the sequence form, $\mathbf{x}_{i+1} = 2 \cdot \mathbf{x}_i \bmod F_c(\mathbf{x})$.

Let $(z_{c-1} \parallel z_{c-2} \parallel \dots \parallel z_1 \parallel z_0)$ denote the state of c -bit LFSR. In our modes, these LFSRs are initialized to $1 \bmod F_c(\mathbf{x})$, i.e., $(0^7 1 \parallel 0^{c-8})$, in little-endian format. Incrementation of LFSRs is defined as follows: for $c = 56$,

$$\begin{aligned} z_i &\leftarrow z_{i-1} \text{ for } i \in \llbracket 56 \rrbracket_0 \setminus \{7, 4, 2, 0\}, \\ z_7 &\leftarrow z_6 \oplus z_{55}, \\ z_4 &\leftarrow z_3 \oplus z_{55}, \\ z_2 &\leftarrow z_1 \oplus z_{55}, \\ z_0 &\leftarrow z_{55}. \end{aligned}$$

7.7. Instantiation of Romulus and Remus with Skinny

Table 7.6.: Members of Romulus.

Family	Name	$\tilde{E}$	k	nl	n	t	d	τ
Romulus-N	Romulus-N1	Skinny-128-384	128	128	128	128	56	128
	Romulus-N2	Skinny-128-384	128	96	128	96	48	128
	Romulus-N3	Skinny-128-256	128	96	128	96	24	128
Romulus-M	Romulus-M1	Skinny-128-384	128	128	128	128	56	128
	Romulus-M2	Skinny-128-384	128	96	128	96	48	128
	Romulus-M3	Skinny-128-256	128	96	128	96	24	128

Similarly, for $c = 24$,

$$\begin{aligned}
z_i &\leftarrow z_{i-1} \text{ for } i \in \llbracket 24 \rrbracket_0 \setminus \{4, 3, 1, 0\}, \\
z_4 &\leftarrow z_3 \oplus z_{23}, \\
z_3 &\leftarrow z_2 \oplus z_{23}, \\
z_1 &\leftarrow z_0 \oplus z_{23}, \\
z_0 &\leftarrow z_{23}.
\end{aligned}$$

Our LFSRs are also called *doubling* over $\text{GF}(2^c)$ in the context of modes [41].

The Tweakkey Encoding

Domain Separation. We will use a domain separation byte B to ensure appropriate independence between the tweakable block cipher calls and the various versions of Romulus. Let $B = (b_7 \| b_6 \| b_5 \| b_4 \| b_3 \| b_2 \| b_1 \| b_0)$ be the bitwise representation of this byte, where b_7 is the MSB and b_0 is the LSB (see also Figure 7.14). Then, we have the following:

- $b_7 b_6 b_5$ will specify the parameter sets. They are fixed to:

- 000 for Romulus-N1
- 001 for Romulus-M1
- 010 for Romulus-N2
- 011 for Romulus-M2
- 100 for Romulus-N3
- 101 for Romulus-M3

Note that all nonce-respecting modes have $b_5 = 0$ and all nonce-misuse resistant modes have $b_5 = 1$.

- b_4 is set to 1 once we have handled the last block of data (AD and message chains are treated separately), to 0 otherwise.
- b_3 is set to 1 when we are performing the authentication phase of the operating mode (i.e., when no ciphertext data is produced), to 0 otherwise. In the special case where $b_5 = 1$ and $b_4 = 1$ (i.e., last block for the nonce-misuse mode), b_3 will instead denote if the number of message blocks is even ($b_5 = 1$ if that is the case, 0 otherwise).

7. Lightweight AEAD from Tweakable Block Ciphers

- b_2 is set to 1 when we are handling a message block, to 0 otherwise. Note that in the case of the misuse-resistant modes, the message blocks will be used during authentication phase (in which case we will have $b_3 = 1$ and $b_2 = 1$). In the special case where $b_5 = 1$ and $b_4 = 1$ (i.e., last block for the nonce-misuse mode), b_3 will instead denote if the number of message blocks is even ($b_5 = 1$ if that is the case, 0 otherwise).
- b_1 is set to 1 when we are handling a padded AD block, to 0 otherwise.
- b_0 is set to 1 when we are handling a padded message block, to 0 otherwise.

The reader can refer to Table 7.7 to obtain the exact specifications of the domain separation values depending on the various cases.

Figure 7.14.: Domain separation when using the tweakable block cipher

Tweakey Encoding. We specify the following tweakey encoding functions for implementing TBC $\tilde{E} : \mathcal{K} \times \overline{\mathcal{T}} \times \mathcal{M} \rightarrow \mathcal{M}$. The tweakey encoding is a function

$$\text{encode}_{m,t} : \mathcal{K} \times \overline{\mathcal{T}} \rightarrow \mathcal{K}_{\mathcal{T}},$$

where $\mathcal{K}_{\mathcal{T}} = \{0, 1\}^m$ is the tweakey space. As defined earlier, $\overline{\mathcal{T}} = \mathcal{T} \times \mathcal{B} \times \mathcal{D}$, $\mathcal{K} = \{0, 1\}^k$ and $\mathcal{T} = \{0, 1\}^t$, $\mathcal{D} = \llbracket 2^d - 1 \rrbracket_0$, $\mathcal{B} = \llbracket 256 \rrbracket_0$.

- Case $(m, t) = (384, 128)$: this variant is used for Romulus-N1 and Romulus-M1. The `encode` function is defined as follows:

$$\text{encode}_{384,128}(K, T, B, D) = \text{lfsr}_{56}(D) \parallel B \parallel 0^{64} \parallel T \parallel K$$

- Case $(m, t) = (384, 96)$: this variant is used for Romulus-N2 and Romulus-M2. The `encode` function is defined as follows:

$$\text{encode}_{384,96}(K, T, B, D) = \text{lfsr}_{24}(D_1) \parallel B \parallel T \parallel K \parallel \text{lfsr}_{24}(D_2) \parallel 0^{104},$$

where $D_1, D_2 \in \mathcal{D}_s$ with $\mathcal{D}_s = \llbracket 2^{24} - 1 \rrbracket$. The set $\mathcal{D}$ is defined as $\mathcal{D} = \mathcal{D}_s \times \mathcal{D}_s$, and the components are determined from D as $D_1 = (D / (2^{24} - 1)) + 1$ and $D_2 = (D \bmod (2^{24} - 1)) + 1$. For the first $2^{24} - 1$ cycles starting from $D = 0$ (but note that $D = 1$ is the initial value in our scheme and $D = 0$ is not used), D_1 is fixed to 1 and D_2 takes all integers of $\llbracket 2^{24} - 1 \rrbracket$. For the next $2^{24} - 1$ cycles, D_1 is fixed to 2 and D_2 takes all values of $\llbracket 2^{24} - 1 \rrbracket$ again, and so on. We stress that the counter cycle is $(2^{24} - 1)^2$ which is slightly smaller than the original range of D . One can also interpret D as a two-dimensional

7.7. Instantiation of Romulus and Remus with Skinny

Table 7.7.: Domain separation byte B of Romulus. Bits b_7 and b_6 are to be set to the appropriate value according to the parameter sets.

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	$\text{int}(B)$	case
Romulus-N	-	-	0	0	1	0	0	0	8	A main
	-	-	0	1	1	0	0	0	24	A last unpadded
	-	-	0	1	1	0	1	0	26	A last padded
	-	-	0	0	0	1	0	0	4	M main
	-	-	0	1	0	1	0	0	20	M last unpadded
	-	-	0	1	0	1	0	1	21	M last padded
Romulus-M	-	-	1	0	1	0	0	0	40	A main
	-	-	1	0	1	1	0	0	44	M auth main
	-	-	1	1	1	1	1	1	63	w: (even,even,padded,padded)
	-	-	1	1	1	1	1	0	62	w: (even,even,padded,unpadded)
	-	-	1	1	1	1	0	1	61	w: (even,even,unpadded,padded)
	-	-	1	1	1	1	0	0	60	w: (even,even,unpadded,unpadded)
	-	-	1	1	1	0	1	1	59	w: (even,odd,padded,padded)
	-	-	1	1	1	0	1	0	58	w: (even,odd,padded,unpadded)
	-	-	1	1	1	0	0	1	57	w: (even,odd,unpadded,padded)
	-	-	1	1	1	0	0	0	56	w: (even,odd,unpadded,unpadded)
	-	-	1	1	0	1	1	1	55	w: (odd,even,padded,padded)
	-	-	1	1	0	1	1	0	54	w: (odd,even,padded,unpadded)
	-	-	1	1	0	1	0	1	53	w: (odd,even,unpadded,padded)
	-	-	1	1	0	1	0	0	52	w: (odd,even,unpadded,unpadded)
	-	-	1	1	0	0	1	1	51	w: (odd,odd,padded,padded)
	-	-	1	1	0	0	1	0	50	w: (odd,odd,padded,unpadded)
	-	-	1	1	0	0	0	1	49	w: (odd,odd,unpadded,padded)
	-	-	1	1	0	0	0	0	48	w: (odd,odd,unpadded,unpadded)
	-	-	1	0	0	1	0	0	36	M enc main

vector: for example, if $D_1 = 0^{23}1$ and $D_2 = 0^{20}1^4$, then $D = (0^{23}1 \parallel 0^{20}1^4)$. In this case, the initial value of D is $[0^7 10^{16}, 0^7 10^{16}]$, in little-endian format.

- Case $(m, t) = (256, 96)$: this variant is used for Romulus-N3 and Romulus-M3. The `encode` function is defined as follows:

$$\text{encode}_{256,96}(K, T, B, D) = \text{lfsr}_{24}(D) \parallel B \parallel T \parallel K$$

For plaintext $M \in \{0, 1\}^n$ and tweak $\overline{T} = (T, B, D) \in \mathcal{T} \times \mathcal{B} \times \mathcal{D}$, $\tilde{E}_K^{(T,B,D)}(M)$ denotes encryption of M with m -bit tweakkey state $\text{encode}_{m,t}(K, T, B, D)$. Tweakey encoding is always implicitly applied, hence the counter D is never arithmetic in the tweakkey state. To avoid confusion, we may write $\overline{D}$ (in particular when it appears in a part of tweak) in order to emphasize that this is indeed an LFSR counter. One can interpret $\overline{D}$ as a state of the LFSR after D clocks (but in that case it is a part of the tweakkey state and *not* a part of input of `encode`).

7.7.2. Instantiating Remus with Skinny

We propose three versions of Remus-N and two versions of Remus-M when instantiated with Skinny. We provide in Table 7.8 the parameters for these variants.

7. Lightweight AEAD from Tweakable Block Ciphers

Table 7.8.: Members of Remus.

Family	Name	E	ICmode	k	nl	n	d	τ
Remus-N	Remus-N1	Skinny-128/128	ICE1	128	128	128	128	128
	Remus-N2	Skinny-128/128	ICE2	128	128	128	128	128
	Remus-N3	Skinny-64/128	ICE3	128	96	64	120	64
Remus-M	Remus-M1	Skinny-128/128	ICE1	128	128	128	128	128
	Remus-M2	Skinny-128/128	ICE2	128	128	128	128	128

Doubling over a Finite Field. For any positive integer c , we assume $\text{GF}(2^c)$ is defined over the lexicographically-first polynomial among the irreducible degree c polynomials of a minimum number of coefficients. We use two fields: $\text{GF}(2^c)$ for $c \in \{128, 120\}$. The primitive polynomials are:

$$x^{128} + x^7 + x^2 + x + 1 \text{ for } c = 128, \quad (7.25)$$

$$x^{120} + x^4 + x^3 + x + 1 \text{ for } c = 120. \quad (7.26)$$

Let $Z = (z_{c-1}z_{c-2} \dots z_1z_0)$ for $z_i \in \{0, 1\}$, $i \in \llbracket c \rrbracket_0$ be an element of $\text{GF}(2^c)$. A multiplication of Z by the generator (polynomial x) is called *doubling* and written as $2Z$ [41]. An i -times doubling of Z is written as 2^iZ , and is efficiently computed from $2^{i-1}Z$ (see below). Here, $2^0Z = Z$ for any Z . When $Z = 0^n$, i.e., zero entity in the field, then $2^iZ = 0^n$ for any $i \geq 0$.

To avoid confusion, we may write $\overline{D}$ (in particular when it appears in a part of tweak) in order to emphasize that this is indeed a doubling-based counter, i.e., $2^D X$ for some key-dependent variable X . One can interpret $\overline{D}$ as 2^D (but in that case it is a part of tweak key state or a coefficient of mask, and *not* a part of input of ICE).

On bit-level, doubling $Z \rightarrow 2Z$ over $\text{GF}(2^c)$ for $c = 128$ is defined as

$$\begin{aligned} z_i &\leftarrow z_{i-1} \text{ for } i \in \llbracket 128 \rrbracket_0 \setminus \{7, 2, 1, 0\}, \\ z_7 &\leftarrow z_6 \oplus z_{127}, \\ z_2 &\leftarrow z_1 \oplus z_{127}, \\ z_1 &\leftarrow z_0 \oplus z_{127}, \\ z_0 &\leftarrow z_{127}. \end{aligned}$$

Similarly, for $\text{GF}(2^{120})$, we have

$$\begin{aligned} z_i &\leftarrow z_{i-1} \text{ for } i \in \llbracket 120 \rrbracket_0 \setminus \{4, 3, 1, 0\}, \\ z_4 &\leftarrow z_3 \oplus z_{119}, \\ z_3 &\leftarrow z_2 \oplus z_{119}, \\ z_1 &\leftarrow z_0 \oplus z_{119}, \\ z_0 &\leftarrow z_{119}. \end{aligned}$$

The TBC ICE

The three variants of ICE are defined as follows. See Section 7.1.5 for reference.

7.7. Instantiation of Romulus and Remus with Skinny

- ICE1: $n = 128$, $nl = 128$, $d = 128$, $k = 128$ and it uses Skinny-128/128 as its building block E .
- ICE2: $n = 128$, $nl = 128$, $d = 128$, $k = 128$ and it uses Skinny-128/128 as its building block E .
- ICE3: $n = 64$, $nl = 96$, $d = 120$, $k = 128$ and it uses Skinny-64/128 as its building block E .

Block Counters and Domain Separation

Domain Separation. We will use a domain separation byte B to ensure appropriate independence between the tweakable block cipher calls and the various versions of Remus. Let $B = (b_7 \| b_6 \| b_5 \| b_4 \| b_3 \| b_2 \| b_1 \| b_0)$ be the bitwise representation of this byte, where b_7 is the MSB and b_0 is the LSB (see also Figure 7.15). Then, we have the following:

- $b_7 b_6 b_5$ will specify the parameter sets. They are fixed to:
 - 000 for Remus-N1
 - 001 for Remus-M1
 - 010 for Remus-N2
 - 011 for Remus-M2
 - 100 for Remus-N3
- Note that all nonce-respecting modes have $b_5 = 0$ and all nonce-misuse resistant modes have $b_5 = 1$.
- b_4 is set to 0.
- b_3 is set to 1 once we have handled the last block of data (AD and message chains are treated separately), to 0 otherwise.
- b_2 is set to 1 when we are performing the authentication phase of the operating mode (i.e., when no ciphertext data is produced), to 0 otherwise. In the special case where $b_5 = 1$ and $b_4 = 1$ (i.e., last block for the nonce-misuse mode), b_3 will instead denote if the number of message blocks is even ($b_5 = 1$ if that is the case, 0 otherwise).
- b_1 is set to 1 when we are handling a message block, to 0 otherwise. Note that in the case of the misuse-resistant modes, the message blocks will be used during authentication phase (in which case we will have $b_3 = 1$ and $b_2 = 1$). In the special case where $b_5 = 1$ and $b_4 = 1$ (i.e., last block for the nonce-misuse mode), b_3 will instead denote if the number of message blocks is even ($b_5 = 1$ if that is the case, 0 otherwise).
- b_0 is set to 1 when we are handling a padded block (associated data or message), to 0 otherwise.

The reader can refer to Table 7.9 to obtain the exact specifications of the domain separation values depending on the various cases.

7. Lightweight AEAD from Tweakable Block Ciphers

Figure 7.15.: Domain separation when using the tweakable block cipher

Table 7.9.: Domain separation byte B of Remus. Bits b_7 and b_6 are to be set to the appropriate value according to the parameter sets.

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	$\text{int}(B)$	case
Remus-N	-	-	0	0	0	1	0	0	4	A main
	-	-	0	0	1	1	0	0	12	A last unpadded
	-	-	0	0	1	1	0	1	13	A last padded
	-	-	0	0	0	0	1	0	2	M main
	-	-	0	0	1	0	1	0	10	M last unpadded
	-	-	0	0	1	0	1	1	11	M last padded
Remus-M	-	-	1	0	0	1	0	0	36	A main
	-	-	1	0	1	1	0	0	44	A last unpadded
	-	-	1	0	1	1	0	1	45	A last padded
	-	-	1	0	0	1	1	0	38	M auth main
	-	-	1	0	1	1	1	0	46	M auth last unpadded
	-	-	1	0	1	1	1	1	47	M auth last padded
	-	-	1	0	0	0	1	0	34	M enc main

Part V.

Conclusions & Further Discussion

8. Conclusion

In this thesis, we studied the topic of designing lightweight hardware-oriented AEAD algorithms from TBCs. We studied the hardware implementation of SKE primitives and the design, implementation and security of TBC-based AEAD algorithms compared to other design approaches.

In Chapter 3 we studied the implementation of cryptanalytic attacks against the SHA-1 hash function. We studied the implementation of both generic birthday search attacks and differential cryptanalytic attacks. We showed that ASIC poses a serious threat for attacks with $O(2^{64})$ complexity. We also showed that birthday attacks with complexity $O(2^{80})$ are not far from being practical. We estimated that it can be implemented in one month for around 62 million USD. We compared the ASIC implementation costs to the GPU implementation costs, describing a trade-off between the two technologies.

In Chapter 4 we implemented two ciphers: AES and LED, describing technology-specific techniques to optimize them for FPGA. We also implemented the Deoxys-I AEAD scheme as an example of Θ CB3 algorithm and studied its properties in hardware.

In Chapter 5 we discussed some of the properties of TBC-based AEAD modes and proposed a metric for estimating their efficiency at early stages of the design.

In Chapter 6 we provided an analysis framework for a group of (T)BC-Based schemes. We applied this framework to COMET-128 and mixFeed. We showed attacks that apply to these designs in the single-user and multi-user settings. For COMET-128, we showed a class of weak keys of size 2^{64} showed a key recovery attack that operates with computational complexity of 2^{65} BC calls, $2^{64}/\mu$ encryption queries and $2^{64}/\mu$ decryption queries, where μ is the number of targeted users. For mixFeed, we identified a weak-key class of high probability ($\sim 45\%$). The existence of such weak key class invalidates the security assumptions made by the designers. The attack proposed was later implemented in practice by Leurent and Pernot [124] using about 220 GBs of data. We also showed an attack against mixFeed in the nonce misuse scenario with negligible data complexity and a single nonce repetition.

In Chapter 7, we proposed to new families of TBC-based AEAD modes: Romulus and Remus. Romulus offers competitive hardware performance with 128-bit security based on the TPRP assumption. It has variants that offer both the NAE (nonce-respecting) and MRAE (misuse-resistant) security notions. Remus provides a security trade-off between the hardware area/state size and the security, where the state size ranges from 256 to 384 bits, while the security ranges from 64 bits to 128 bits. Its security is based on the ideal cipher model. The Remus family also includes NAE and MRAE variants. We studied the hardware implementation of these designs, describing different implementations that we have tested practically.

9. Other Works

Between 2016 and 2020, I have participated in multiple projects that are not part of this thesis. These projects cover a wide range of topics related to symmetric key cryptography. In this chapter, I list some of these projects.

Differential Fault Attacks on SPNs and Countermeasures. In [142], we provide a generic DFA attack method targeting Substitution-Permutation Network (SPN) based families of symmetric block ciphers. We show that for any SPN, as long as the fault mask injected before a non-linear layer in the last round follows a non-uniform distribution, the key search space can always be reduced. This shows that it is not possible to design an SPN-based cipher that is completely secure against DFA, without randomization. Furthermore, we propose a novel approach to find good fault masks that can leak the key with a small number of instances. We then developed a tool, called *Joint Difference Distribution Table (JDDT)* for pre-computing the solutions for the fault equations, which allows us to recover the last round key with a very small number of pairs of faulty and non-faulty ciphertexts. We evaluate our methodology on various block ciphers, including PRESENT-80, PRESENT-128, GIFT-64, GIFT-128, AES-128, LED-64, LED-128, SKINNY-64-64, SKINNY-128-128, PRIDE and PRINCE.

In [143], we extend this attack to a set of AEAD modes that use SPNs as an underlying primitive. We study the challenges imposed by the design of these modes, such as masking of the cipher-text. We also show that a very small number of nonce repetition and faults is required to extend the original attack on AES, which makes it very practical. We show that OCB and COLM need 1 nonce repetition and 3 faults only to uniquely identify the Key.

In [144], we provide a method to incorporate an implementation-based countermeasure, in the form of a LASER sensor, into any ASIC circuit. In [145], we give a high-level countermeasure that provides a safe-guard against DFA for block ciphers, at a low cost.

Countermeasure for Statistical Ineffective Fault Attacks. State-of-the-art countermeasures against Fault Injection Attacks (FIA) provide good protection against analysis methods that require the differences in the correct and faulty ciphertext to derive the secret information, such as Differential Fault Analysis (DFA) or collision fault analysis. However, recent progress in Ineffective Fault Analysis (IFA) and Statistical IFA (SIFA) constitutes a real threat against cryptographic implementations. Such methods cannot be thwarted by standard FIA countermeasures that focus on detecting the change in the intermediate data. In [146], we present a novel method based on error correcting codes that protects implementations against SIFA. We design a set of universal error-correcting gates that can be used for block cipher implementations. We analyze a hardware implementation of protected GIFT-64 and show that our method provides 100% protection against SIFA. In [147], we characterize the strategies used in some of the early SIFA

9. Other Works

countermeasures and study the cost of implementing each of them. We also study the security advantages and disadvantages of each of them and show some of the shortcomings of these countermeasures.

Analysis of Lightweight AEAD Algorithms. In addition to the AEAD algorithms described in this thesis, I have participated in the analysis of two other NIST lightweight cryptography candidates: SNEIK [148] and Orange [149].

Hardware Implementation of Boolean Functions. In [150], we investigate the hardware circuit complexity of the class of Boolean functions recently introduced by Tang and Maitra [151]. While this class of functions has very good cryptographic properties, the exact hardware requirement is an immediate concern as noted in the paper itself. In this direction, we consider different circuit architectures based on finite field arithmetic and Boolean optimization. An estimation of the circuit complexity is provided for such functions given any input size n . We study different candidate architectures for implementing these functions, all based on the finite field arithmetic. We also show different implementations for both ASIC and FPGA, providing further analysis on the practical aspects of the functions in question and the relation between these implementations and the theoretical bound.

10. Future Work

The research results achieved through-out this thesis have opened the door to more research problems to be addressed. In the near future, I plan to work on some of these ideas. Two of these ideas are listed in this chapter.

Leakage-resilient AEAD from TBCs. The topic of leakage-resilient constructions is a growing topic of research. It refers to constructions that offer provable security against side-channel attacks in the presence of leakage, at the algorithmic level. I plan to study the use of TBCs in such constructions in more details.

ASIC Benchmarking of the NIST lightweight cryptography candidates. In collaboration with several researchers, we have recently announced an ASIC benchmarking project for the NIST lightweight cryptography candidates. I plan to study the performance and implementation cost of these candidates using different technologies and provide some insights into the trade-offs they offer for ASIC implementations.

Bibliography

- [1] Frank Carter. *The Turing Bombe*. Bletchley Park Trust, 2008. http://www.rutherfordjournal.org/article030108.html?&sa=U&ei=rHEhVKewDMbRywOHq4HoAQ&ved=0CDwQFjAJ&usq=AFQjCNEN34UrBDkFM6vnW-bX_fUGzXDCjw.
- [2] Brian Randell. Colossus: Godfather of the Computer. In *The Origins of Digital Computers*, pages 349–354. Springer, 1982. https://link.springer.com/chapter/10.1007/978-3-642-61812-3_27.
- [3] NIST. Submission Requirements and Evaluation Criteria for the Lightweight Cryptography Standardization Process, 2018. <https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/final-lwc-submission-requirements-august2018.pdf>.
- [4] CAESAR Competition. CAESAR submissions. <https://competitions.cr.yp.to/caesar-submissions.html>, 2020.
- [5] Hongjun Wu. Acorn: a Lightweight Authenticated Cipher v3. CAESAR competition, 2016. <https://competitions.cr.yp.to/round3/acornv3.pdf>.
- [6] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schlffer. Ascon v1.2. CAESAR competition, 2019. <https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/ascon-spec-round2.pdf>.
- [7] David A. McGrew and John Viega. The Security and Performance of the Galois/Counter Mode (GCM) of Operation. In Anne Canteaut and Kapalee Viswanathan, editors, *Progress in Cryptology - INDOCRYPT 2004*, volume 3348 of *LNCS*, pages 343–355. Springer, 2004. https://link.springer.com/chapter/10.1007/978-3-540-30556-9_27.
- [8] Phillip Rogaway, Mihir Bellare, and John Black. OCB: A Block-Cipher Mode of Operation for Efficient Authenticated Encryption. *ACM Transaction on Information and Systems Security*, 6(3):365–403, 2003. <https://dl.acm.org/doi/10.1145/937527.937529>.
- [9] Moses Liskov, Ronald L. Rivest, and David Wagner. Tweakable Block Ciphers. In Moti Yung, editor, *Advances in Cryptology — CRYPTO 2002*, pages 31–46, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. https://link.springer.com/chapter/10.1007/3-540-45708-9_3.
- [10] Ted Krovetz and Phillip Rogaway. The Software Performance of Authenticated-Encryption Modes. In Antoine Joux, editor, *Fast Software*

Bibliography

- Encryption*, pages 306–327, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. https://link.springer.com/chapter/10.1007/978-3-642-21702-9_18.
- [11] Thomas Peyrin and Yannick Seurin. Counter-in-Tweak: Authenticated Encryption Modes for Tweakable Block Ciphers. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology – CRYPTO 2016*, pages 33–63, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. https://link.springer.com/chapter/10.1007/978-3-662-53018-4_2.
- [12] Meltem Sönmez Turan, Kerry A McKay, Çağdaş Çalk, Donghoon Chang, and Larry Bassham. Status Report on the First Round of the NIST Lightweight Cryptography Standardization Process. Technical report, 2019. <https://www.nist.gov/publications/status-report-first-round-nist-lightweight-cryptography-standardization-process>.
- [13] The Merriam-Webster Dictionary, 2020. <https://www.merriam-webster.com/dictionary/cryptography>.
- [14] Mustafa Khairallah, Zakaria Najm, Anupam Chattopadhyay, and Thomas Peyrin. Crack Me if You Can: Hardware Acceleration Bridging the Gap Between Practical and Theoretical Cryptanalysis?: A Survey. In *Proceedings of the 18th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, SAMOS '18*, pages 167–172, New York, NY, USA, 2018. ACM. <http://doi.acm.org/10.1145/3229631.3239366>.
- [15] George Boole. *The Mathematical Analysis of Logic*. Philosophical Library, 1847. <https://books.google.com.sg/books?id=zv4YAQAIAAJ>.
- [16] C. E. Shannon. Communication Theory of Secrecy Systems. *The Bell System Technical Journal*, 28(4):656–715, 1949. <https://ieeexplore.ieee.org/abstract/document/6769090>.
- [17] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: The advanced encryption standard (AES)*. Springer Science & Business Media, 2013. <https://link.springer.com/book/10.1007%2F978-3-662-60769-5>.
- [18] Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. The SKINNY family of block ciphers and its low-latency variant MANTIS. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology – CRYPTO 2016*, pages 123–153, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. https://link.springer.com/chapter/10.1007/978-3-662-53008-5_5.
- [19] Jian Guo, Thomas Peyrin, Axel Poschmann, and Matt Robshaw. The LED Block Cipher. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2011*, pages 326–341, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. https://link.springer.com/chapter/10.1007/978-3-642-23951-9_22.

- [20] Jérémy Jean, Ivica Nikolić, and Thomas Peyrin. Tweaks and Keys for Block Ciphers: The TWEAKEY Framework. In Palash Sarkar and Tetsu Iwata, editors, *Advances in Cryptology – ASIACRYPT 2014*, pages 274–288, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg. https://link.springer.com/chapter/10.1007/978-3-662-45608-8_15.
- [21] Jérémy Jean, Ivica Nikolic, Thomas Peyrin, and Yannick Seurin. Deoxys v1.41. *CAESAR competition*, 2016. <https://competitions.cr.yp.to/round3/deoxysv14.pdf>.
- [22] Jérémy Jean, Amir Moradi, Thomas Peyrin, and Pascal Sasdrich. Bit-Sliding: A Generic Technique for Bit-Serial Implementations of SPN-based Primitives. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, pages 687–707, Cham, 2017. Springer International Publishing. https://link.springer.com/chapter/10.1007/978-3-319-66787-4_33.
- [23] Amir Moradi, Axel Poschmann, San Ling, Christof Paar, and Huaxiong Wang. Pushing the Limits: A Very Compact and a Threshold Implementation of AES. In Kenneth G. Paterson, editor, *Advances in Cryptology – EUROCRYPT 2011*, pages 69–88, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. https://link.springer.com/chapter/10.1007/978-3-642-20465-4_6.
- [24] Subhadeep Banik, Andrey Bogdanov, and Francesco Regazzoni. Atomic-AES: A Compact Implementation of the AES Encryption/Decryption Core. In Orr Dunkelman and Somitra Kumar Sanadhya, editors, *Progress in Cryptology – INDOCRYPT 2016*, pages 173–190, Cham, 2016. Springer International Publishing. https://link.springer.com/chapter/10.1007/978-3-319-49890-4_10.
- [25] Subhadeep Banik, Andrey Bogdanov, and Francesco Regazzoni. Atomic-AES v2.0. *IACR Cryptology ePrint Archive, Report 2016/1005*, 2016. <https://eprint.iacr.org/2016/1005>.
- [26] NIST: National Institute of Standards and Technology. Advanced Encryption Standard AES, 2001. <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf>.
- [27] Ralph Charles Merkle. *Secrecy, Authentication, and Public Key Systems*. Computer science: Systems programming. UMI Research Press, 1982. <https://books.google.com.sg/books?id=24C8wgEACAAJ>.
- [28] Marc Stevens, Arjen K Lenstra, and Benne De Weger. Chosen-Prefix Collisions for MD5 and Applications. *International Journal of Applied Cryptography*, 2(4):322–359, 2012. <https://www.inderscienceonline.com/doi/abs/10.1504/IJACT.2012.048084>.
- [29] Xiaoyun Wang, Andrew C Yao, and Frances Yao. Cryptanalysis on SHA-1. In *Cryptographic Hash Workshop hosted by NIST*, 2005. https://csrc.nist.gov/groups/ST/hash/documents/Wang_SHA1-New-Result.pdf.

- [30] Muhammad Hassan, Ayesha Khalid, Anupam Chattopadhyay, Christian Rechberger, Tim Güneysu, and Christof Paar. New ASIC/FPGA Cost Estimates for SHA-1 Collisions. In *2015 Euromicro Conference on Digital System Design*, pages 669–676, 2015. <https://ieeexplore.ieee.org/abstract/document/7302342>.
- [31] Marc Stevens. New Collision Attacks on SHA-1 Based on Optimal Joint Local-Collision Analysis. In *Advances in Cryptology – EUROCRYPT 2013*, pages 245–261, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. https://link.springer.com/chapter/10.1007/978-3-642-38348-9_15.
- [32] Gaëtan Leurent and Thomas Peyrin. From Collisions to Chosen-Prefix Collisions Application to Full SHA-1. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019*, pages 527–555, Cham, 2019. Springer International Publishing. https://link.springer.com/chapter/10.1007/978-3-030-17659-4_18.
- [33] Gaëtan Leurent and Thomas Peyrin. SHA-1 is a Shambles - First Chosen-Prefix Collision on SHA-1 and Application to the PGP Web of Trust. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1839–1856, 2020. <https://eprint.iacr.org/2020/014>.
- [34] Paul C Van Oorschot and Michael J Wiener. Parallel Collision Search with Cryptanalytic Applications. *Journal of cryptology*, 12(1):1–28, 1999. <https://doi.org/10.1007/PL00003816>.
- [35] John M Pollard. Monte Carlo methods for Index Computation (mod p). *Mathematics of computation*, 32(143):918–924, 1978. <https://www.ams.org/journals/mcom/1978-32-143/S0025-5718-1978-0491431-9/S0025-5718-1978-0491431-9.pdf>.
- [36] Wikimedia Commons. See https://commons.wikimedia.org/wiki/File:Ecb_mode_pic.png.
- [37] Wikimedia Commons. See https://commons.wikimedia.org/wiki/File:No_ecb_mode_picture.png.
- [38] Mihir Bellare and Chanathip Namprempre. Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm. In Tatsuaki Okamoto, editor, *Advances in Cryptology — ASIACRYPT 2000*, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. https://link.springer.com/chapter/10.1007/3-540-44448-3_41.
- [39] Phillip Rogaway. Authenticated-Encryption with Associated-Data. In *Proceedings of the 9th ACM Conference on Computer and Communications Security, CCS '02*, page 98–107, New York, NY, USA, 2002. Association for Computing Machinery. <https://dl.acm.org/doi/abs/10.1145/586110.586125>.
- [40] Mihir Bellare, Phillip Rogaway, and David Wagner. The EAX Mode of Operation. In Bimal Roy and Willi Meier, editors, *Fast Software Encryption*, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. https://link.springer.com/chapter/10.1007/978-3-540-25937-4_25.

- [41] Phillip Rogaway. Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC. In Pil Joong Lee, editor, *Advances in Cryptology - ASIACRYPT 2004*, pages 16–31, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. https://link.springer.com/chapter/10.1007/978-3-540-30539-2_2.
- [42] Mihir Bellare and Chanathip Namprempre. Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm. pages 531–545, 2000. https://link.springer.com/chapter/10.1007/3-540-44448-3_41.
- [43] Phillip Rogaway. Nonce-Based Symmetric Encryption. In Bimal Roy and Willi Meier, editors, *Fast Software Encryption*, pages 348–358, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. https://link.springer.com/chapter/10.1007/978-3-540-25937-4_22.
- [44] Phillip Rogaway and Thomas Shrimpton. A Provable-Security Treatment of the Key-Wrap Problem. In Serge Vaudenay, editor, *Advances in Cryptology - EUROCRYPT 2006*, pages 373–390, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. https://link.springer.com/chapter/10.1007/11761679_23.
- [45] Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. Skinny-AEAD and Skinny-Hash. *NIST Lightweight Cryptography Project*, 2020. <https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/SKINNY-spec-round2.pdf>.
- [46] Avik Chakraborti, Tetsu Iwata, Kazuhiko Minematsu, and Mridul Nandi. Blockcipher-Based Authenticated Encryption: How Small Can We Go? In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems – CHES 2017*, pages 277–298, Cham, 2017. Springer International Publishing. https://link.springer.com/chapter/10.1007/978-3-319-66787-4_14.
- [47] Subhadeep Banik, Avik Chakraborti, Tetsu Iwata, Kazuhiko Minematsu, Mridul Nandi, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. GIFT-COFB. *NIST Lightweight Cryptography Project*, 2020. <https://csrc.nist.gov/Projects/Lightweight-Cryptography/Round-1-Candidates>.
- [48] Ethash. *GitHub Ethereum Wiki*, 2017. <https://github.com/ethereum/wiki/wiki/Ethash>.
- [49] X16R. <https://en.bitcoinwiki.org/wiki/X16R>.
- [50] Sandeep Kumar, Christof Paar, Jan Pelzl, Gerd Pfeiffer, and Manfred Schimmler. Breaking ciphers with COPACOBANA—a cost-optimized parallel code breaker. In Louis Goubin and Mitsuru Matsui, editors, *Cryptographic Hardware and Embedded Systems - CHES 2006*, pages 101–118, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. https://link.springer.com/chapter/10.1007/11894063_9.

Bibliography

- [51] Christopher Swenson. *Modern Cryptanalysis: Techniques for Advanced Code Breaking*. John Wiley & Sons, 2008. <https://books.google.com.sg/books?id=oLoaWgdmFJ8C>.
- [52] Antoine Joux. *Algorithmic Cryptanalysis*. CRC Press, 2009. https://books.google.com.sg/books?id=buQajqt-_iUC.
- [53] Mark Stamp and Richard M Low. *Applied Cryptanalysis: Breaking Ciphers in the Real World*. John Wiley & Sons, 2007. <https://books.google.com.sg/books?id=buVGyPNbwJUC>.
- [54] Whitfield Diffie and Martin E Hellman. Special Feature Exhaustive Cryptanalysis of the NBS Data Encryption Standard. *Computer*, 10(6):74–84, 1977. <https://ieeexplore.ieee.org/abstract/document/1646525>.
- [55] Martin Hellman. A Cryptanalytic Time-Memory Trade-off. *IEEE transactions on Information Theory*, 26(4):401–406, 1980. <https://ieeexplore.ieee.org/abstract/document/1056220>.
- [56] John Gilmore. Cracking DES: Secrets of Encryption Research, Wiretap Politics & Chip Design, 1998. <https://dl.acm.org/doi/abs/10.5555/551916>.
- [57] Tim Güneysu, Timo Kasper, Martin Novotný, Christof Paar, and Andy Rupp. Cryptanalysis with COPACOBANA. *IEEE Transactions on Computers*, 57(11):1498–1513, 2008. <https://ieeexplore.ieee.org/abstract/document/4515858/>.
- [58] Jörg Keller. A Hardware-Based Attack on the A5/1 Stream Cipher. 2001. <https://pdfs.semanticscholar.org/d7f9/e63574bdab7d412ffc8fad79711533140459.pdf>.
- [59] Sam Biddle. NYU Accidentally Exposed Military Code-Breaking Computer Project To Entire Internet. *The Intercept*, 2017. <https://theintercept.com/2017/05/11/nyu-accidentally-exposed-military-code-breaking-computer-project-to-entire-internet/>.
- [60] Eran Tromer. *Hardware-Based cryptanalysis*. 2007. <https://www.cs.tau.ac.il/~tromer/phd-dissertation/>.
- [61] Vikram Suresh, Sudhir Satpathy, and Sanu Mathew. Bitcoin Mining Hardware Accelerator with Optimized Message Digest and Message Scheduler Datapath, 2018. WIPO WO2018057282A1 <https://patents.google.com/patent/WO2018057282A1/en>.
- [62] Nick Marinoff. Samsung is Building ASIC Chips for Halong Mining. *The Bitcoin Magazine*, 2018. <https://bitcoinmagazine.com/articles/samsung-building-asic-chips-halong-mining/>.
- [63] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The First Collision for Full SHA-1. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017*, pages 570–596, Cham, 2017. Springer International Publishing. https://link.springer.com/chapter/10.1007/978-3-319-63688-7_19.

- [64] Karthikeyan Bhargavan and Gaëtan Leurent. Transcript Collision Attacks: Breaking Authentication in TLS, IKE and SSH. In *Network and Distributed System Security Symposium - NDSS*, 2016. <https://hal.inria.fr/hal-01244855/document>.
- [65] Andrey Bogdanov, Dmitry Khovratovich, and Christian Rechberger. Biclique Cryptanalysis of the Full AES. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology – ASIACRYPT 2011*, pages 344–371, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. https://link.springer.com/chapter/10.1007/978-3-642-25385-0_19.
- [66] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK Lightweight Block Ciphers. In *Proceedings of the 52nd Annual Design Automation Conference, DAC '15*, New York, NY, USA, 2015. Association for Computing Machinery. <https://dl.acm.org/doi/abs/10.1145/2744769.2747946>.
- [67] Tom Brostöm. Lightweight Trusted Computing. *NIST Lightweight Cryptography Workshop - Invited Talk*, 2019. <https://www.nist.gov/news-events/events/2019/11/lightweight-cryptography-workshop-2019>.
- [68] Pierre Karpman, Thomas Peyrin, and Marc Stevens. Practical Free-Start Collision Attacks on 76-step SHA-1. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology – CRYPTO 2015*, pages 623–642, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. https://link.springer.com/chapter/10.1007/978-3-662-47989-6_30.
- [69] Marc Stevens, Pierre Karpman, and Thomas Peyrin. Freestart Collision for full SHA-1. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, pages 459–483, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. https://link.springer.com/chapter/10.1007/978-3-662-49890-3_18.
- [70] Antoine Joux and Thomas Peyrin. Hash Functions and the (Amplified) Boomerang Attack. In Alfred Menezes, editor, *Advances in Cryptology – CRYPTO 2007*, pages 244–263, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. https://link.springer.com/chapter/10.1007/978-3-540-74143-5_14.
- [71] Siva Narendra, Matthew Haycock, Venkatesh Govindarajulu, Vasantha Erraguntla, Howard Wilson, Sriram Vangal, Amaresh Pangal, Erik Seligman, Raj Nair, Ali Keshavarzi, Bradley Bloechel, Gregory Dermer, Randy Mooney, Nitin Borkar, Shekhar Borkar, and Vivek De. 1.1 V 1 GHz Communications Router with On-Chip Body Bias in 150 nm CMOS. In *2002 IEEE International Solid-State Circuits Conference. Digest of Technical Papers (Cat. No.02CH37315)*, volume 1, pages 270–466 vol.1, 2002. <https://ieeexplore.ieee.org/remotexs.ntu.edu.sg/stamp/stamp.jsp?tp=&arnumber=993040>.
- [72] Ying-Mei Tu and Chun-Wei Lu. The Influence of Lot Size on Production Performance in Wafer Fabrication Based on Simulation. In *Procedia Engineering*,

Bibliography

- volume 174, pages 135 – 144, 2017. <http://www.sciencedirect.com/science/article/pii/S1877705817301807>.
- [73] Handel Jones. FINFET and FD-SOI: Market and Cost Analysis. *FDSOI Forum 2018*, 2018. <http://soiconsortium.eu/wp-content/uploads/2018/08/MS-FDSOI9.1818-cr.pdf>.
- [74] Global Petrol Prices. <https://www.globalpetrolprices.com>.
- [75] Mustafa Khairallah, Anupam Chattopadhyay, and Thomas Peyrin. Looting the LUTs: FPGA optimization of AES and AES-like ciphers for authenticated encryption. In Arpita Patra and Nigel P. Smart, editors, *Progress in Cryptology – INDOCRYPT 2017*, pages 282–301, Cham, 2017. Springer International Publishing. https://link.springer.com/chapter/10.1007/978-3-319-71667-1_15.
- [76] Niels Ferguson, Stefan Lucks, Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas, and Jesse Walker. The Skein Hash Function Family. *Submission to the NIST Hash Function Standardization Process (round 3)*, 7(7.5):3, 2010. <http://www.skein-hash.info/sites/default/files/skein1.3.pdf>.
- [77] John Black and Phillip Rogaway. A Block-Cipher Mode of Operation for Parallelizable Message Authentication. In Lars R. Knudsen, editor, *Advances in Cryptology — EUROCRYPT 2002*, pages 384–397, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. https://link.springer.com/chapter/10.1007/3-540-46035-7_25.
- [78] Kazuhiko Minematsu. AES-OTR v3.1. *CAESAR competition*, 2016. <https://competitions.cr.ypt.to/round3/aesotrv31.pdf>.
- [79] Russell Housley. Using Advanced Encryption Standard (AES) Counter Mode with IPSEC Encapsulating Security Payload (ESP). Technical report, RFC 3686, January, 2004. <https://dl.acm.org/doi/book/10.117487/RFC3686>.
- [80] CERG: Cryptographic Engineering Research Group, George Mason University. ATHENa: Automated Tools for Hardware EvaluationN. <https://cryptography.gmu.edu/athena/>, 2017.
- [81] Ekawat "Ice" Homsirikamol. Deoxys-I-128 Implementation. <https://cryptography.gmu.edu/athena/>, 2016.
- [82] Ekawat Homsirikamol, William Diehl, Ahmed Ferozpuri, Farnoud Farahmand, Panasayya Yalla, Jens-Peter Kaps, and Kris Gaj. CAESAR Hardware API. *IACR Cryptology ePrint Archive, Report 2016/626*, 2016. <https://eprint.iacr.org/2016/626>.
- [83] Axel Poschmann and Marc Stöttinger. Deoxys-I-128 Implementation. <https://cryptography.gmu.edu/athena/>, 2016.
- [84] João Carlos Resende and Ricardo Chaves. AES Datapaths on FPGAs: A State of the Art Analysis. In Nicolas Sklavos, Ricardo Chaves, Giorgio Di Natale, and Francesco Regazzoni, editors, *Hardware Security and Trust:*

- Design and Deployment of Integrated Circuits in a Threatened Environment*, pages 1–25. Springer International Publishing, Cham, 2017. https://link.springer.com/chapter/10.1007/978-3-319-44318-8_1.
- [85] Qiang Liu, Zhenyu Xu, and Ye Yuan. A 66.1 Gbps single-pipeline AES on FPGA. In *2013 International Conference on Field-Programmable Technology (FPT)*, pages 378–381, Dec 2013. <https://ieeexplore.ieee.org/abstract/document/6718392/>.
 - [86] Mazen El Maraghy, Salma Hesham, and Mohamed A Abd El Ghany. Real-Time Efficient FPGA Implementation of AES Algorithm. In *2013 IEEE International SOC Conference*, pages 203–208, 2013. <https://ieeexplore.ieee.org/document/6749688>.
 - [87] Ricardo Chaves, Georgi Kuzmanov, Stamatis Vassiliadis, and Leonel Sousa. Reconfigurable Memory based AES co-processor. In *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, 2006. <https://ieeexplore.ieee.org/document/1639441>.
 - [88] Philippe Bulens, François-Xavier Standaert, Jean-Jacques Quisquater, Pascal Pellegrin, and Gaël Rouvroy. Implementation of the AES-128 on Virtex-5 FPGAs. In Serge Vaudenay, editor, *Progress in Cryptology – AFRICACRYPT 2008*, pages 16–26, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. https://link.springer.com/chapter/10.1007/978-3-540-68164-9_2.
 - [89] Joan Boyar and René Peralta. A New Combinational Logic Minimization Technique with Applications to Cryptology. In Paola Festa, editor, *Experimental Algorithms*, pages 178–189, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. https://link.springer.com/chapter/10.1007/978-3-642-13193-6_16.
 - [90] Joan Boyar and René Peralta. A Small Depth-16 Circuit for the AES S-Box. In Dimitris Gritzalis, Steven Furnell, and Marianthi Theoharidou, editors, *Information Security and Privacy Research*, pages 287–298, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. https://link.springer.com/chapter/10.1007/978-3-642-30436-1_24.
 - [91] D. Canright. A Very Compact S-Box for AES. In Josyula R. Rao and Berk Sunar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2005*, pages 441–455, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg. https://link.springer.com/chapter/10.1007/11545262_32.
 - [92] Alexander Maximov. AES MixColumn with 92 XOR gates. *IACR Cryptology ePrint Archive, Report 2019/833*. <https://eprint.iacr.org/2019/833>.
 - [93] Solmaz Ghaznavi, Catherine Gebotys, and Reouven Elbaz. Efficient Technique for the FPGA Implementation of the AES MixColumns Transformation. In *Reconfigurable Computing and FPGAs, 2009. ReConFig'09. International Conference on*, pages 219–224. IEEE, 2009. <https://ieeexplore.ieee.org/abstract/document/5382055>.

Bibliography

- [94] Joao Carlos Resende and Ricardo Chaves. Compact Dual Block AES Core on FPGA for CCM Protocol. In *Field Programmable Logic and Applications (FPL), 2015 25th International Conference on*, pages 1–8. IEEE, 2015. <https://ieeexplore.ieee.org/document/7293948/>.
- [95] N. Nalla Anandakumar, Thomas Peyrin, and Axel Poschmann. A Very Compact FPGA Implementation of LED and PHOTON. In Willi Meier and Debdeep Mukhopadhyay, editors, *Progress in Cryptology – INDOCRYPT 2014*, pages 304–321, Cham, 2014. Springer International Publishing. https://link.springer.com/chapter/10.1007/978-3-319-13039-2_18.
- [96] Rich Schroepel and Hilarie Orman. The Hasty Pudding Cipher. *NIST AES Submission*, 1998. <http://www.princeton.edu/~rblee/HPC/>.
- [97] Kosei Sakamoto, Kazuhiko Minematsu, Nao Shibata, Maki Shigeri, Hiroyasu Kubo, Yuki Funabiki, Andrey Bogdanov, Sumio Morioka, and Takanori Isobe. Tweakable TWINE: Building a Tweakable Block Cipher on Generalized Feistel Structure. In Nuttapong Attrapadung and Takeshi Yagi, editors, *Advances in Information and Computer Security*, pages 129–145, Cham, 2019. Springer International Publishing. https://link.springer.com/chapter/10.1007/978-3-030-26834-3_8.
- [98] Tomoyasu Suzaki, Kazuhiko Minematsu, Sumio Morioka, and Eita Kobayashi. TWINE: A Lightweight Block Cipher for Multiple Platforms. In Lars R. Knudsen and Huapeng Wu, editors, *Selected Areas in Cryptography*, pages 339–354, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. https://link.springer.com/chapter/10.1007/978-3-642-35999-6_22.
- [99] Niels Ferguson. Collision Attacks on OCB. 2002. <https://csrc.nist.gov/csrc/media/projects/block-cipher-techniques/documents/bcm/comments/general-comments/papers/ferguson.pdf>.
- [100] Alexandre Adomnicai, Zakaria Najm, and Thomas Peyrin. Fixslicing: A New GIFT Representation: Fast Constant-Time Implementations of GIFT and GIFT-COFB on ARM Cortex-M. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(3):402–427, Jun. 2020. <https://tches.iacr.org/index.php/TCHES/article/view/8595>.
- [101] Tetsu Iwata, Mustafa Khairallah, Kazuhiko Minematsu, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Ling Sun. TGIF v1, 2019. <https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/round-1/spec-doc/TGIF-spec.pdf>.
- [102] Yusuke Naito, Yu Sasaki, and Takeshi Sugawara. Lightweight Authenticated Encryption Mode Suitable for Threshold Implementation. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology – EUROCRYPT 2020*, pages 705–735, Cham, 2020. Springer International Publishing. https://link.springer.com/chapter/10.1007/978-3-030-45724-2_24.

- [103] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015*, pages 430–454, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. https://link.springer.com/chapter/10.1007/978-3-662-46800-5_17.
- [104] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Duplexing the Sponge: Single-Pass Authenticated Encryption and Other Applications. In Ali Miri and Serge Vaudenay, editors, *Selected Areas in Cryptography*, pages 320–337, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. https://link.springer.com/chapter/10.1007/978-3-642-28496-0_19.
- [105] Avik Chakraborti, Nilanjan Datta, Mridul Nandi, and Kan Yasuda. Beetle Family of Lightweight and Secure Authenticated Encryption Ciphers. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(2):218–241, May 2018. <https://tches.iacr.org/index.php/TCHES/article/view/881>.
- [106] Tim Beyne, Yu Long Chen, Christoph Dobraunig, and Bart Mennink. Elephant. NIST Lightweight Cryptography Project, 2020. <https://csrc.nist.gov/Projects/Lightweight-Cryptography/round-2-candidates>.
- [107] Orr Dunkelman, Nathan Keller, and Adi Shamir. Minimalism in Cryptography: The Even-Mansour Scheme Revisited. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology – EUROCRYPT 2012*, pages 336–354, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. https://link.springer.com/chapter/10.1007/978-3-642-29011-4_21.
- [108] Tetsu Iwata. New Blockcipher Modes of Operation with Beyond the Birthday Bound Security. In Matthew Robshaw, editor, *Fast Software Encryption*, pages 310–327, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. https://link.springer.com/chapter/10.1007/11799313_20.
- [109] Yusuke Naito and Takeshi Sugawara. Lightweight Authenticated Encryption Mode of Operation for Tweakable Block Ciphers. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(1):66–94, Nov. 2019. <https://tches.iacr.org/index.php/TCHES/article/view/8393>.
- [110] Davide Bellizia, Francesco Berti, Olivier Bronchain, Gaëtan Cassiers, Sébastien Duval, Chun Guo, Gregor Leander, Gaëtan Leurent, Itamar Levi, Charles Momin, Olivier Pereira, Thomas Peters, François-Xavier Standaert, and Friedrich Wiemer. Spook. NIST Lightweight Cryptography Project, 2020. <https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/Spook-spec-round2.pdf>.
- [111] Francesco Berti, Chun Guo, Olivier Pereira, Thomas Peters, and François-Xavier Standaert. TEDT, a Leakage-Resist AEAD Mode for High Physical

Bibliography

- Security Applications. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020(1):256–320, Nov. 2019. <https://tches.iacr.org/index.php/TCHES/article/view/8400>.
- [112] Tetsu Iwata, Mustafa Khairallah, Kazuhiko Minematsu, and Thomas Peyrin. Romulus v1.2. NIST Lightweight Cryptography Project, 2020. <https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/Romulus-spec-round2.pdf>.
- [113] Tetsu Iwata, Kazuhiko Minematsu, Thomas Peyrin, and Yannick Seurin. ZMAC: A Fast Tweakable Block Cipher Mode for Highly Secure Message Authentication. In *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part III*, pages 34–65, 2017. https://doi.org/10.1007/978-3-319-63697-9_2.
- [114] Tetsu Iwata, Mustafa Khairallah, Kazuhiko Minematsu, and Thomas Peyrin. Remus v1, 2019. <https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/round-1/spec-doc/Remus-spec.pdf>.
- [115] Subhadeep Banik, Andrey Bogdanov, Atul Luykx, and Elmar Tischhauser. SUNDAE: Small Universal Deterministic Authenticated Encryption for the Internet of Things. *IACR Transactions on Symmetric Cryptology*, 2018(3):1–35, Sep. 2018. <https://tosc.iacr.org/index.php/ToSC/article/view/7296>.
- [116] Mustafa Khairallah. Weak Keys in the Rekeying Paradigm: Application to COMET and mixFeed. *IACR Transactions on Symmetric Cryptology*, 2019(4):272–289, Jan. 2020. <https://tosc.iacr.org/index.php/ToSC/article/view/8465>.
- [117] Mustafa Khairallah. Forgery Attack on mixFeed in the Nonce-Misuse Scenario. *IACR Cryptology ePrint Archive, Report 2019/457*. <https://eprint.iacr.org/2019/457.pdf>.
- [118] Tetsu Iwata, Mustafa Khairallah, Kazuhiko Minematsu, and Thomas Peyrin. Duel of the Titans: The Romulus and Remus Families of Lightweight AEAD Algorithms. *IACR Transactions on Symmetric Cryptology*, 2020(1):43–120, May 2020. <https://tosc.iacr.org/index.php/ToSC/article/view/8560>.
- [119] Avik Chakraborti, Nilanjan Datta, Ashwin Jha, and Mridul Nandi. HyENA. NIST Lightweight Cryptography Project, 2020. <https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/hyena-spec-round2.pdf>.
- [120] Shay Gueron, Ashwin Jha, and Mridul Nandi. COMET: COunter Mode Encryption with authentication Tag. NIST Lightweight Cryptography Project, 2020. <https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/comet-spec-round2.pdf>.

- [121] Bishwajit Chakraborty and Mridul Nandi. mixFeed. NIST Lightweight Cryptography Project, 2020. <https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/mixFeed-spec-round2.pdf>.
- [122] Atul Luykx, Bart Mennink, and Kenneth G. Paterson. Analyzing Multi-key Security Degradation. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017*, pages 575–605, Cham, 2017. Springer International Publishing. https://link.springer.com/chapter/10.1007/978-3-319-70697-9_20.
- [123] Shay Gueron, Ashwin Jha, and Mridul Nandi. On the Security of COMET Authenticated Encryption Scheme. *NIST Lightweight Cryptography Workshop*, 2019. <https://csrc.nist.gov/CSRC/media/Events/lightweight-cryptography-workshop-2019/documents/papers/on-the-security-of-comet-lwc2019.pdf>.
- [124] Gaëtan Leurent and Clara Pernot. A New Representation of the AES-128 Key Schedule: Application to mixFeed and ALE. Private Communication, 2020.
- [125] Helena Handschuh and Bart Preneel. Key-recovery attacks on universal hash function based MAC algorithms. In David Wagner, editor, *Advances in Cryptology – CRYPTO 2008*, pages 144–161, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. https://link.springer.com/chapter/10.1007/978-3-540-85174-5_9.
- [126] Mohamed Ahmed Abdelraheem, Andrey Bogdanov, and Elmar Tischhauser. Weak-Key Analysis of POET. *IACR Cryptology ePrint Archive, Report 2014/226*. <https://eprint.iacr.org/2014/226.pdf>.
- [127] Mihir Bellare and Björn Tackmann. The Multi-user Security of Authenticated Encryption: AES-GCM in TLS 1.3. In Matthew Robshaw and Jonathan Katz, editors, *Advances in Cryptology – CRYPTO 2016*, pages 247–276, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. https://link.springer.com/chapter/10.1007/978-3-662-53018-4_10.
- [128] Ran Canetti, Oded Goldreich, and Shai Halevi. The Random Oracle Methodology, Revisited. In *Journal of the ACM*, pages 209–218. ACM, 2004. <https://dl.acm.org/doi/abs/10.1145/1008731.1008734>.
- [129] Bart Mennink. Optimally Secure Tweakable Blockciphers. In Gregor Leander, editor, *Fast Software Encryption*, pages 428–448, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. https://link.springer.com/chapter/10.1007/978-3-662-48116-5_21.
- [130] Nicky Mouha, Bart Mennink, Anthony Van Herrewege, Dai Watanabe, Bart Preneel, and Ingrid Verbauwhede. Chaskey: An Efficient MAC Algorithm for 32-bit Microcontrollers. In Antoine Joux and Amr Youssef, editors, *Selected Areas in Cryptography – SAC 2014*, pages 306–323, Cham, 2014. Springer International Publishing. https://link.springer.com/chapter/10.1007/978-3-319-13051-4_19.

Bibliography

- [131] Donghoon Chang, Morris Dworkin, Seokhie Hong, John Kelsey, and Mridul Nandi. A Keyed Sponge Construction with Pseudo-randomness in the Standard Model. *NIST SHA-3 2012 Workshop*, 2012. http://csrc.nist.gov/groups/ST/hash/sha-3/Round3/March2012/documents/papers/CHANG_paper.pdf.
- [132] Bart Mennink, Reza Reyhanitabar, and Damian Vizár. Security of Full-State Keyed Sponge and Duplex: Applications to Authenticated Encryption. In Tetsu Iwata and Jung Hee Cheon, editors, *Advances in Cryptology – ASIACRYPT 2015*, pages 465–489, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. https://link.springer.com/chapter/10.1007/978-3-662-48800-3_19.
- [133] Nilanjan Datta, Ashwin Jha, Alexandre Meje, and Mridul Nandi. Breaking REMUS and TGIF in the light of NIST Lightweight Cryptography Standardization Project. NIST Lightweight Cryptography Workshop, 2019. <https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/round-1/official-comments/REMUS-official-comment.pdf>.
- [134] Ashwin Jha, Eik List, Kazuhiko Minematsu, Sweta Mishra, and Mridul Nandi. XHX - A Framework for Optimally Secure Tweakable Block Ciphers from Classical Block Ciphers and Universal Hashing, 2019. https://link.springer.com/chapter/10.1007/978-3-030-25283-0_12.
- [135] Christophe De Cannière, Orr Dunkelman, and Miroslav Knežević. Katan and ktantan — a family of small and efficient hardware-oriented block ciphers. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009*, pages 272–288, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
- [136] Hannes Groß, Erich Wenger, Christoph Dobraunig, and Christoph Ehrenhöfer. Suit up!—Made-to-Measure Hardware Implementations of ASCON. In *2015 Euromicro Conference on Digital System Design*, pages 645–652. IEEE, 2015. <https://ieeexplore.ieee.org/abstract/document/7302339>.
- [137] Sachin Kumar, Jawad Haj-Yihia, Mustafa Khairallah, and Anupam Chattopadhyay. A Comprehensive Performance Analysis of Hardware Implementations of CAESAR Candidates. *IACR Cryptology ePrint Archive, Report 2017/1261*. <https://eprint.iacr.org/2017/1261.pdf>.
- [138] Ryad Benadjila, Jian Guo, Victor Lomné, and Thomas Peyrin. Implementing Lightweight Block Ciphers on x86 Architectures. In *Selected Areas in Cryptography - SAC 2013 - 20th International Conference, Burnaby, BC, Canada, August 14-16, 2013, Revised Selected Papers*, pages 324–351, 2013. <https://link.springer.com/book/10.1007%2F978-3-662-43414-7>.
- [139] Andrey Bogdanov, Martin M. Lauridsen, and Elmar Tischhauser. Comb to Pipeline: Fast Software Encryption Revisited. In *Fast Software Encryption - 22nd International Workshop, FSE 2015, Istanbul, Turkey, March 8-11, 2015, Revised Selected Papers*, pages 150–171, 2015. https://link.springer.com/chapter/10.1007%2F978-3-662-48116-5_8.

- [140] Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. The Skinny Cipher Website. <https://sites.google.com/site/skinnycipher/home>.
- [141] Eric Wenger and Hannes Groß. ASCON-128 Hardware Design Document, 2017. https://github.com/IAIK/ascon_hardware/blob/master/doc/ascon_hw_doc.pdf.
- [142] Mustafa Khairallah, Xiaolu Hou, Zakaria Najm, Jakub Breier, Shivam Bhasin and Thomas Peyrin. SoK: On DFA Vulnerabilities of Substitution-Permutation Networks. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, Asia CCS '19, page 403–414, New York, NY, USA, 2019. Association for Computing Machinery. <https://dl.acm.org/doi/abs/10.1145/3321705.3329810>.
- [143] Mustafa Khairallah, Shivam Bhasin, Anupam Chattopadhyay. On Misuse of Nonce-Misuse Resistance : Adapting Differential Fault Attacks on (few) CAESAR Winners. In *2019 IEEE 8th International Workshop on Advances in Sensors and Interfaces (IWASI)*, pages 189–193, June 2019. <https://ieeexplore.ieee.org/abstract/document/8791393/>.
- [144] Mustafa Khairallah, Rajat Sadhukhan, Radhamanjari Samanta, Jakub Breier, Shivam Bhasin, Rajat S. Chakraborty, Anupam Chattopadhyay and Debdeep Mukhopadhyay. DFARPA: Differential fault attack resistant physical design automation. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1171–1174, March 2018. <https://ieeexplore.ieee.org/document/8342190>.
- [145] Anubhab Baksi, Shivam Bhasin, Jakub Breier, Mustafa Khairallah and Thomas Peyrin. Protecting block ciphers against differential fault attacks without re-keying. In *2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 191–194, 2018. <https://ieeexplore.ieee.org/document/8383913>. Full version: <https://eprint.iacr.org/2018/085>.
- [146] Jakub Breier, Mustafa Khairallah, Xialou Hou and Yang Liu. A Countermeasure Against Statistical Ineffective Fault Analysis. *IEEE Transactions on Circuits and Systems II: Express Briefs*, pages 1–1, 2020. <https://ieeexplore.ieee.org/document/9075200>.
- [147] Mustafa Khairallah, Shivam Bhasin and Karim M. Abdellatif. On Comparison of Countermeasures against Statistical Ineffective Fault Attacks. In *2019 31st International Conference on Microelectronics (ICM)*, pages 122–125, 2019. <https://ieeexplore.ieee.org/abstract/document/9021933>.
- [148] Mustafa Khairallah. Forgery Attack on SNEIKEN. *Cryptology ePrint Archive, Report 2019/408*. <https://eprint.iacr.org/2019/408>.
- [149] Sumanta Sarkar, Mustafa Khairallah and Raghvendra Rohit. Round 2 Official Comment: ORANGE. NIST Lighweight Cryptography Forum, 2019. <https://csrc.nist.gov/CSRC/media/Projects/>

Bibliography

lightweight-cryptography/documents/round-2/official-comments/orange-round2-official-comment.pdf.

- [150] Mustafa Khairallah, Anupam Chattopadhyay, Bimal Mandal, and Subhamoy Maitra. On Hardware Implementation of Tang-Maitra Boolean Functions. In Lilya Budaghyan and Francisco Rodríguez-Henríquez, editors, *Arithmetic of Finite Fields*, pages 111–127, Cham, 2018. Springer International Publishing. https://link.springer.com/chapter/10.1007/978-3-030-05153-2_6.
- [151] Deng Tang and Subhamoy Maitra. Construction of n -variable ($n \equiv 2 \pmod{4}$) balanced boolean functions with maximum absolute value in autocorrelation spectra $< 2^{\frac{n}{2}}$. *Cryptology ePrint Archive, Report 2016/1078*. <https://eprint.iacr.org/2016/1078>.