UNIVERSITÉ DU LUXEMBOURG

PhD-FSTM-2022-082
The Faculty of Science, Technology and Medicine

DISSERTATION

Defence held on 24/06/2022 in Esch-sur-Alzette

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG

EN INFORMATIQUE

by

Luan CARDOSO DOS SANTOS
Born on 19 January 1993 in Pompéia - São Paulo, Brazil

DESIGN, CRYPTANALYSIS AND PROTECTION OF SYMMETRIC ENCRYPTION ALGORITHMS

Dissertation defence committee

Dr Alex Biryukov, dissertation supervisor
Professor, Université du Luxembourg

Dr. Jean-Sébastien Coron, Chairman
Professor, Université du Luxembourg

Dr. Volker Müller, Vice Chairman
Associate Professor, Université du Luxembourg

Dr. Diego Aranha
Associate Professor, Aarhus University, Denmark

Dr.techn. Maria Eichlsede
Associate Professor, Graz University of Technology, Austria

"*Life is a journey. Time is a River. The door is ajar.*"

Jim Butcher

Abstract

This thesis covers results from several areas related to symmetric cryptography, secure and efficient implementation and is divided into four main parts:

In Part II, Benchmarking of AEAD, two articles will be presented, showing the results of the FELICS framework for Authenticated encryption algorithms, and multi-architecture benchmarking of permutations used as construction block of AEAD algorithms.

The Sparkle family of Hash and AEAD algorithms will be shown in Part III. Sparkle is currently a finalist of the NIST call for standardization of lightweight hash and AEAD algorithms.

In Part IV, Cryptanalysis of ARX ciphers, it is discussed two cryptanalysis techniques based on differential trails, applied to ARX ciphers. The first technique, called Meet-in-the-Filter uses an offline trail record, combined with a fixed trail and a reverse differential search to propose long differential trails that are useful for key recovery. The second technique is an extension of ARX analyzing tools, that can automate the generation of truncated trails from existing non-truncated ones, and compute the exact probability of those truncated trails.

In Part V, Masked AES for Microcontrollers, is shown a new method to efficiently compute a side-channel protected AES, based on the masking scheme described by Rivain and Prouff. This method introduces table and execution-order optimizations, as well as practical security proofs.

Acknowledgements

We all walk a path, from the moment we are born down to the day we are no more. The beauty of this path is that, while it is unique to every person, we share parts of it with others. Sometimes there is a lot of people together, sometimes just a few. There are people who just crosses paths with us, and there are those we share the longest of walks with. This path of life is sometimes rough, difficult, dangerous, but you can always find a hand willing to help you continue forward. I am very thankful to the people that I have met, that helped me go forward. To name them all, I would need way more than a page. But even if names are not written, I am thankful and honored to have walked side by side with you.

To my family –Enezina, Gabriel, Bruno– for always being by my side and always being a source of comfort.

To the Trautsch family, for adopting me, showing love as they would to a son, and being my little safe haven, my home away from home.

To my friends, always supportive, without whom my life would be less fun. And thanks, specially, for all the times you patiently listened to me ramble.

To my professors, for teaching and guiding me.

To my work colleagues, for their helpfulness and professionalism.

A special thanks to Bill, Fahl, and Zé. You guys are better friends than I deserve.

A very special thank you to Michael. I would not be without you.

Contents

Part I

Introduction

Chapter 1

Introduction

1.1 Introduction to Cryptography

This section presents a small introduction to the concepts relevant to this dissertation, for a reader that is not well acquainted with the area of cryptography.

Starting with cryptography, it is an area of both mathematics and computer science that studies techniques that are older than the Roman Republic, whose last leader gave name to both a cryptography technique and the forthcoming emperors – Cæsar. Those techniques came to be by the necessity of *securing information*, keeping it away from prying eyes, from military orders to *coup d'etat*, from conspirations of death to letters of forbidden love. The name cryptography –and cryptology– comes from the Greek words "kryptós", meaning "hidden, secret", "graphein" meaning "to write", and "logia" meaning "study": Cryptography is the study of hidden and secret writing. In a general manner, cryptography is an area of study that deals with constructing and analyzing tools that protect data in the presence of a malicious adversary, be it data in transit, at rest, or even during processing.

During most of its history, cryptography lacked rigor in its construction and usage: cryptographers and cryptanalysts relied more on their instincts to secure messages. At this time, the used techniques were fairly simple. For example, many techniques were simply different forms of *substitution cipher* where symbols were simply replaced by others, following a simple look-up, akin to taking a text in the Latin alphabet and writing it in the Greek one. Other ciphers could be reduced to permutation ciphers, where the symbols were just rearranged following a simple rule, for example, writing a message in a grid as row-major and sending it "encrypted" as column-major. It was during the second world war that those techniques were improved greatly. There were huge advantages in keeping the enemy from reading your messages, as well as being able to read the enemies', and quickly cipher machines were greatly improved, together with techniques and machines to defeat those. The most well-known of those machines was the Enigma, used by the German military. The efforts to decrypt Enigma messages were a substantial aid to the Allied war effort and laid the groundwork for not only modern cryptography but also for the development of general-use electronic computers.

Nowadays, cryptography in general is ubiquitous in our lives, and in many situations, transparent. Innumerable hours of work and research are used, unnoticed, whenever we send an email, access a website over HTTPS, or use a credit card for online shopping. Even more, computation is not restricted to desk devices. Most people carry in their pockets computers that would look like science fiction just a handful of years ago; Doors, cars, watches, sensors, and furniture, are all interconnected and need efficient and secure ways of protecting their data. Not only a tool of war anymore, the study of cryptography is the study of tools that help groups, and individuals, remain safe.

As an academic discipline, research in this area is relatively recent, beginning in the mid-1970s. It was at that time that IBM designed the Data Encryption Standard (DES) algorithm, which was adopted by the USA government as a standard. By the end of that decade, the Diffie-Hellman key exchange algorithm was published, as well as the RSA algorithm. Modern cryptography can be divided into many ways, with two very important groups being Symmetric-key cryptography and Public-key cryptography. In the following sections, we will discuss some of the relevant subareas of Symmetric cryptography, as well as other important topics for this thesis.

1.1.1 Symmetric cryptography and block ciphers

One of the big areas of study in cryptography is the so-called *Symmetric Cryptography*, also known as *secret key cryptography*. Symmetric-key algorithms are those that use the same key for both encryption and decryption procedures, with those keys being a shared secret between the involved parties. Practically, the encryption and decryption keys are not required to be identical, but a simple transformation can be used to convert one into another. The complementing part of symmetric cryptography is the public-key cryptography, where a pair of keys, one public and another private, are used and the private one cannot be easily derived from the public one.

The requirement that a secret is shared between parties might be seen as a disadvantage of symmetric algorithms, but this is balanced by its usefulness in encrypting large amounts of data, its small key size, and its highspeed. Matter of fact, in situations where one would expect to use a public key encryption algorithm, in most cases the protocol will use it for secret exchange, and then resort to a symmetric algorithm for bulk data encryption.

Symmetric encryption algorithms can be either stream ciphers or block ciphers. In a stream cipher, the plaintext is combined with a pseudorandom keystream, where each plaintext bit is encrypted one at a time with the keystream, generating a ciphertext stream. A block cipher, on the other hand, is an algorithm that operates over a fixed length of input data, the so-called *block*. The common method for designing block ciphers consists of iterating transformations that combine permutation and substitution transformations. This design based on the iteration of simple transformations was proposed by Shannon in 1949 [Sha49]. One of the first public block ciphers was the DES (FIPS 46-3), published by the U.S. National Bureau of Standards, and later superseded by AES, also by NIST[1], via a public competition. Block ciphers are of great importance in cryptography, as they are not only used to encrypt data (using a mode of operation to encrypt more data than the length of a block), but they are used as building blocks in many other cryptographic algorithms, such as stream ciphers, hash functions, pseudo-random number generators, message authentication codes, and authenticated encryption.

[1]The USA's National Institute of Standards and Technology –NIST– was called the National Bureau of Standards –NBS– from its foundation in 1901 to 1988. The Data Encryption Standard –DES– was published as an official Federal Information Processing Standard –FIPS– by the NBS in 1977 as FIPS PUB 46, with the third and last revision being published in 1999.

1.1.2 Authenticated encryption

Authenticated Encryption(AE) and Authenticated Encryption with Associated Data (AEAD) are symmetric ciphers able to simultaneously provide confidentiality and authenticity of data. Such a scheme is useful, for example, to encrypt the body of an email, and have the header information as associated data. In this case, the body of the email will be confidential, while reader information is kept as plaintext, to allow the correct routing of the message, and the whole is authenticated, guaranteeing that it was not manipulated in transit.

The need for authenticated encryption algorithms started with the fact that combining separated encryption and authentication algorithms is non-trivial and error-prone, exemplified by practical attacks against widely used protocols such as the SSL/TLS. The first standardization of AE(AD) was by the creation of six AE modes of operation, in ISO/IEC 19772:2009. Latter, a competition called CAESAR was announced, which encouraged the creation of new AEAD algorithms, not restricted to modes of operation over a block cipher and MAC, but also featuring dedicated designs. Currently, NIST opened a competition for the design and standardization of a family of Lightweight Authenticated encryption algorithms.

1.1.3 Lightweight cryptography

A recent concept in the cryptography area, *lightweight cryptography* is a research area driven by the lack of proper primitives for usage on constrained and low-power devices. Devices such as RFID tags, sensors, IoT devices, wearable computers, and others, are often constrained in terms of available energy and hardware, and the security-efficiency tradeoffs of current cryptographic algorithms are not appropriate, hence the need for specific solutions. The concept of lightweightness is often measured against both hardware and software constraints: In software, small usage of RAM and ROM is desirable, as well as factors such as throughput and latency, the latter especially desirable in real-time applications, where processing agility is needed. In the realm of hardware implementations, the area a cryptographic function takes on the chip is usually as important as its speed. For both cases, power consumption is also very important, as many of those constrained devices operate with limited power supplies, such as batteries, and in some cases, energy harvested from their surroundings, for example, smart-home controllers that use photovoltaic panels.

1.1.4 NIST and cryptography competitions

The National Institute of Standards and Technology (NIST) is a science laboratory and non-regulatory agency in the USA. The core competencies of NIST are measurement science, traceability, and the development and use of standards. In the area of Cryptography, NIST was responsible for standardizing cryptographic algorithms that are today in widespread use, many of them chosen via cryptographic competitions.

Cryptographic competitions are public calls for designs of cryptographic algorithms, which are submitted and analyzed both by private organizations and academics. The main objective of those competitions is to inspire cryptanalysis efforts and, ultimately, choose one or more standards.

The first open cryptographic competition started in January of 1997, held by NIST, and had the objective of selecting a successor to the DES block cipher, which resulted in one of today's most used algorithm, the *Advanced Encryption Standard*, or

AES. This competition received 15 competitors, and after intensive analysis by the public, NIST, and other competitors, Rijndael was chosen among five finalists. One important characteristic of the AES competition, and other competitions, was that the candidates had strong motivations for analyzing and finding flaws in their competitors. The end result is a group of motivated cryptographers checking each other's work, which increases the confidence in the security of the winner.

After the success of the AES competition, several others followed with different objectives. Some of them are:

- NESSIE: The *New European Schemes for Signatures, Integrity and Encryption* was a research project to identify secure cryptographic primitives, from 2000 to 2003. NESSIE selected block ciphers, MAC, Hash functions, public key encryption, identification schemes, and digital signature algorithms. The competition received 42 candidates and selected twelve algorithms, plus five publicly known ones that were not submitted explicitly but considered nonetheless. The project stated that "no weaknesses were found in the selected designs".

- CRYPTREC: The *Cryptography Research and Evaluation Committees* were set up by the Japanese government to select, evaluate, and recommend cryptography for both government and industrial use. It is comparable to the EU's NESSIE and USA's AES competition. One characteristic of CRYPTREC was its obligation to take into account previously existing standards and practices. This led to some unusual features, such as recommending several block ciphers with 64-bit keys, and the inclusion of 160-bit hash functions, which should not be used in new systems designs.

- eSTREAM: The *ECRYPT Stream Cipher Project*, as the name suggests, had the objective of promoting the design of efficient stream ciphers suitable for widespread use. The project ran from 2004 to 2008, and in its latest review, published a portfolio of seven ciphers divided into two profiles: A software-oriented profile, suitable for software applications with high throughput requirements; and a hardware profile, adequate for applications with restricted resources like limited storage, gate count, or power consumption.

- SHA-3: The *NIST hash function competition*, sometimes called the SHA-3 competition, was held by NIST to develop a new hash function to complement the older –and broken– SHA-1, and SHA-2. The competition was held from 2007 to 2012 and announced Keccak as the hash function to be published as NIST FIPS 202 "SHA-3 Standard", which complements FIPS 180-4 *Secure Hash Standard.*

- PHC: The *Password Hashing Competition*, announced in 2013, was an open competition to select password hash functions that could be recommended as standards. Modeled after the NIST competitions, PHC was directly organized by cryptographers to raise awareness of the need for strong password hashing algorithms and was motivated by a series of breaches in popular services, such as the ones that targeted the Playstation Network(2011), LinkedIn, Adobe, ASUS(2012). Among the 24 received candidates, in 2015 Argon2 was chosen as the recommended algorithm and is currently accredited by entities such as IETF and OWASP (resp. *Internet Engineering Task Force* and *Open Web Application Security Project*).

- CAESAR: The *Competition for Authenticated Encryption: Security, Applicability, and Robustness* had the objective of selecting a portfolio of authenticated encryption algorithms adequate for widespread usage and which would offer advantages over AES-GCM. It started in 2014, and the final portfolio was announced in early 2019. CAESAR's portfolio is divided into three use cases: Lightweight applications for constrained environments (Ascon and ACORN), high-performance applications (AEGIS and OCB), and defense in depth (Deoxys-II and COLM); two ciphers were selected for each use case.

- Chinese Cryptographic Algorithm Design Competition[2]: Sponsored by the SCA (Chinese State Cryptography Administration) was a competition held from June 2018 to December 2019, with the objective of implementing the strategic strengthening of the country through the internet, promoting the design of cryptographic algorithms and the growth of cryptographic talents. The competition had 60 submissions, being 22 block ciphers and 32 public-key algorithms, and finished with the identification of the three best public-key algorithms (Aigis-sig, LAC.PKE, and Aigis-enc) and the two best block ciphers (uBlock and Ballet).
- PQC: in 2016, NIST started the *Post-Quantum Cryptography Standardization Process*, to solicit, evaluate, and standardize new public-key cryptography standards, which are capable of protecting sensitive information in the foreseeable future, including after the full realization of quantum computers. This competition is still ongoing and received 82 submissions, of which there are seven third-round finalists and 8 alternate candidates. It is expected that the draft standard will be posted for public comments in 2022-2023.

Following the footsteps of the block cipher, hash function, stream cipher, and other competitions, NIST announced the Lightweight Cryptography Standardization Process. The project was initiated in 2015 after NIST identified that the performance of NIST-approved cryptographic standards (especially AEAD and hash functions) on constrained devices was not adequate. As of 2021, out of the 57 submitted algorithms, ten remain finalists, namely ASCON, Elephant, GIFT-COFB, Grain128-AEAD, ISAP, Photon-Beetle, Romulus, Sparkle, TinyJambu, and Xoodyak. This final round of the standardization process is expected to conclude at the end of 2022. Sparkle is part of this thesis and will be discussed in Chapter 4.

1.1.5 Cryptanalysis

Cryptanalysis is the process of analyzing systems with the objective of understanding hidden aspects of the systems and is used to breach cryptographic systems. The goal of cryptanalysis is to gain as much information as possible about the plaintext via the ciphertext, which is classified by the amount of information available to the attacker:

- Ciphertext only, where the attacker has access only to a set of ciphertexts,
- Known-Plaintext, where the attacker has access to a set of ciphertexts and their corresponding plaintexts.
- Chosen-plaintext and Chosen-Ciphertext, where the attacker can choose the plaintext/ciphertexts and get their corresponding encryptions/decryptions. This type of attack can also be adaptive, where the attacker is allowed to choose the next data based on information gathered from the previous data.

[2]Original document in `www.cacrnet.org.cn/site/content/854.html`. Accessed 27.05.2022.

- Related-key attack, where the attacker has access to ciphertexts encrypted under different unknown keys but related in a known manner, e.g. they differ in a single bit.

Those attacks can be further characterized by the resources needed to carry out the attack (time, memory, and data).

1.1.5.1 Differential cryptanalysis

Differential cryptanalysis is a form of cryptanalytic technique, mainly applicable to block ciphers, which studies how differences in information input can affect the differences in the outputs, where difference is normally the XOR of the pair of plaintexts/-ciphertexts. The academic discovery of differential cryptanalysis is due to Biham and Shamir in the late 80s, in their seminal work on cryptanalysis of DES [BS91]. Even though they managed to find an academic attack on DES which required 2^{47} effort and chosen plaintexts vs. 2^{56} exhaustive search complexity, they found that the cipher was unexpectedly resistant to this powerful attack.

In 1994 Don Coppersmith, a member of the original IBM team that designed DES stated that the technique was already known by both IBM and NSA, which choose to not make the design considerations (and the attack, by extension) public, as that would weaken competitive advantages the USA had concerning cryptographic techniques.

In a simple manner, a differential cryptanalysis attack starts with the attacker choosing plaintexts related by a given difference. The attacker then tries to predict the ciphertext differences, in an effort to discover non-random statistical patterns. Such prediction is typically done by tracing the difference propagation through the rounds of a cipher forming the so-called *differential trail* whose probability (under certain independence assumptions) can be computed as a product of round transition probabilities. Since for a statistical distinguisher typically only input and output differences matter, several trails which have common inputs/outputs can be combined into so-called *differential*, thus accumulating higher distinguishing probability. Key recovery is, for example, done by the attacker requesting a large number of plaintext pairs, and assuming that the difference holds for r-1 rounds. The attacker filters away wrong keys which do not lead to the difference predicted by the differential distinguisher. The correct key is then recovered by trial encryption with the candidate keys.

1.1.6 Side Channel leakage and countermeasures

In addition to the mathematical analysis of the algorithms, cryptanalysis also deals with side-channel attacks, that don't target the algorithm itself but instead target weakness in the implementation of algorithms. Side-channel attacks exploit timing, power consumption, electromagnetic emanations, and other sources to gather the information that can be exploited.

A power analysis attack uses intimate knowledge of the implementation and platform to correlate the power consumption with the secret data being processed. For example, exponentiation is used in many cryptosystems and is often implemented via the square-and-multiply algorithm. By observing the power consumption of the chip during this exponentiation, one is capable to differentiate a bit 0 and 1, since a set bit will execute an extra multiplication. This type of direct interpretation of the power traces during a cryptographic operation is called SPA – simple power analysis.

Beyond SPA, a more advanced attack is the Differential Power Analysis (DPA), which is a statistical method to analyze side-channel measurement sets to identify data-dependent non-random statistical biases. DPA is more powerful than SPA because even tiny correlations can be seen regardless of noise, given enough measurements. Furthermore, DPA can treat the cipher implementation as a black box, by exploring correlations between the measurements and a model of the power consumption of the system (sometimes as simple as using the Hamming weight of the private data).

Defenses against side-channel are dependent on the type of leakage, and the architecture. One approach is to combine the secret information with random noise, called masking, where the data is decoupled from the system's biases. This combination of data and randomness works in a way analogue to Shamir's secret sharing, which itself is a generalization of the one-time pad. This technique, though, needs intimate knowledge of the target system, as structures transparent to the programmer (for example, temporary registers, pipelines, and speculative execution) can inadvertently recombine masked values, causing them to leak.

1.1.7 The ARM architecture

ARM –Advanced RISC Machine– is a popular architecture developed by the British company ARM Holdings. A RISC processor is, in comparison to CISC, simpler to design and requires fewer transistors in their design. ARM is one of the most popular processor architectures in the world, in terms of manufacture number, being used in small sensors, real-time applications, cars, consumer electronics, and even in servers. The majority of ARM processors support fixed-length 32-bit instructions, and mixed 16- and 32-bit instructions for code density. The newer specification of the architecture, ARMv8, supports 64-bit addressing space and arithmetic. The main features of the ARM instruction set are:

- Uniform 32-bit or 64-bit registers.
- Single-cycle instructions (mostly) as well conditional execution for those instructions.
- A zero-penalty barrel shifter, which permits to execute arithmetic and shift operations in a single instruction
- A load/store-based programming model, where the instructions are divided into two main groups: Memory access instructions and logic/arithmetic instructions. Differently from a *register memory architecture*, all the instructions operands must reside in registers.

1.1.8 Other remarks

This thesis is written in a modified "collection of papers" style, instead of a single coherent monograph, as it better represents the style of work done during the Doctoral studies. Each chapter follows a paper and contains most of the information from the original publication. As those were group efforts, the papers might have sections where I had low input in the research. Those are not present in this thesis, and the reader is invited to read the full papers for those. This thesis goes over different but complementary areas: Work was done on the benchmarking of authenticated encryption algorithms, on the development of one such algorithm, then cryptanalytical techniques based on differential trails, and finally, the masking of AES.

1.2 Research papers

The scientific production during the Doctoral studies is as follow:

Publications

[Bei+19] Christof Beierle, Alex Biryukov, Luan Cardoso dos Santos, Johann Großschädl, Léo Perrin, Aleksei Udovenko, Vesselin Velichkov, and Qingju Wang. *Schwaemm and Esch: Lightweight Authenticated Encryption and Hashing using the Sparkle Permutation Family.* `https://www.cryptolux.org/index.php/Sparkle`. 2019.

[Bei+20a] Christof Beierle, Alex Biryukov, Luan Cardoso dos Santos, Johann Großschädl, Léo Perrin, Aleksei Udovenko, Vesselin Velichkov, and Qingju Wang. "Alzette: A 64-Bit ARX-box". In: *Advances in Cryptology – CRYPTO 2020.* Ed. by Daniele Micciancio and Thomas Ristenpart. Cham: Springer International Publishing, 2020, pp. 419–448. ISBN: 978-3-030-56877-1.

[Bei+20d] Christof Beierle, Alex Biryukov, Luan Cardoso dos Santos, Johann Großschädl, Léo Perrin, Aleksei Udovenko, Vesselin Velichkov, and Qingju Wang. "Lightweight AEAD and hashing using the sparkle permutation family". In: *IACR Transactions on Symmetric Cryptology* (2020), pp. 208–261.

[Bir+22] Alex Biryukov, Luan Cardoso dos Santos, Daniel Feher, Vesselin Velichkov, and Giuseppe Vitto. "Automated Truncation of Differential Trails and Trail Clustering in ARX". In: *Selected Areas in Cryptography.* Ed. by Riham AlTawy and Andreas Hülsing. Cham: Springer International Publishing, 2022, pp. 286–307. ISBN: 978-3-030-99277-4.

[SG21] Luan Cardoso dos Santos and Johann Großschädl. "An Evaluation of the Multi-Platform Efficiency of Lightweight Cryptographic Permutations". In: *Innovative Security Solutions for Information Technology and Communications* (2021).

[SGB19] Luan Cardoso dos Santos, Johann Großschädl, and Alex Biryukov. "FELICS-AEAD: benchmarking of lightweight authenticated encryption algorithms". In: *International Conference on Smart Card Research and Advanced Applications.* Springer. 2019, pp. 216–233.

Unrefereed Publications

[Ale+22] Biryukov Alex, Luan Cardoso dos Santos, Je Sen Teh, Aleksei Udovenko, and Vesselin Velichkov. "Meet-in-the-Filter and Dynamic Counting with Applications to Speck". Cryptology ePrint Archive, Paper 2022/673. In submission. 2022. URL: `https://eprint.iacr.org/2022/673/`.

[San+22] Luan Cardoso dos Santos, François Gérard, Johann Großschädl, and Lorenzo Spignoli. "Rivain-Prouff on Steroids: Faster and Stronger Masking of the AES". In submission. 2022.

Furthermore, results regarding FELICS and the benchmarks were presented in the following workshops:

- **Lightweight Cryptography Workshop 2019**: On November 04-06, 2019, presentation on the FELICS framework [SGB19].

- **Lightweight Cryptography Workshop 2020**: On October 19-21, 2020, presentation on the Multiplataform efficiency of the LWC permutations [SG21].

1.3 Thesis structure

This thesis is organized analogous to the execution timeline of the doctoral studies, and it divided into four parts and it is organized as following:

In Part II, titled "Benchmarking of AEAD", I will present the work done on extending the FELICS framework for bechmarking AEAD algorithms, and also the results of specific benchmark of AEAD primitives, with focus to the NIST lightweight AEAD competition. This section is based on two publications [SGB19] and [SG21].

In Part III, I will present my contributions to the family of AEAD ciphersSPARKLE. Creating an AEAD family of ciphers is no trivial task, and as such, SPARKLEis a joint effort of many scientists, with different and complementary skillsets; and at the time of the writing of this thesis, a sucessful effort, as SPARKLE is a finalist of the NIST competition. My focus is on implementation aspects, and how the design of the cipher was guided to be efficient in microcontrollers. This part is based on the SPARKLE publications, and for a full description of the algorithms, the reader is invited to consult [Bei+19; Bei+20a; Bei+20d].

In Part IV –Cryptanalysis of ARX ciphers– two cryptanalysis techniques will be presented. First, based on [Ale+22], is a differential-based attack against ARX ciphers, that combines a online offline parts to generate long trail guesses for ciphertext and plaintext pairs to be used in a key recovery procedure. This attack was instantiated against the block cipher SPECK. Also will be discussed the results from [Bir+22], where an efficient tool for automated truncation of differential trails in ARX ciphers is proposed.

Lastly, in Part V, based on [San+22], is shown an improved method for masking AES, that results in a very fast and secure first-order masked implementation. In this Part, I also discuss a discrepancy between the practical and theoretical aspects of side-channel protection papers, where, until now, very few papers execute practical leakage tests, and when tested, show very relevant levels of leakage.

Part II

Benchmarking of AEAD

Chapter 2

FELICS Framework

In this section, I present an adaptation of the paper titled “FELICS-AED: Benchmarking of Lightweight authenticated encryption algorithms"[SGB19]. This paper was published in the *International Conference on Smart Card Research and Advanced Applications*, and was also featured in the NIST Lightweight Cryptography Workshop 2019.

Cryptographic algorithms that can simultaneously provide both encryption and authentication play an increasingly important role in modern security architectures and protocols (e.g. TLS v1.3). Dozens of authenticated encryption systems have been designed in the past years, which has initiated a large body of research in cryptanalysis. The interest in authenticated encryption has further risen after the National Institute of Standards and Technology (NIST) announced an initiative to standardize “lightweight” authenticated ciphers and hash functions that are suitable for resource-constrained devices. However, while there already exist some cryptanalytic results on these recent designs, information is lacking regarding their performance, especially when they are executed on small 8, 16, and 32-bit microcontrollers. In the follwoing paper, it is introduced an open-source benchmarking tool suite for a fair and consistent evaluation of Authenticated Encryption with Associated Data (AEAD) algorithms written in C or assembly language for 8-bit AVR, 16-bit MSP430, and 32-bit ARM Cortex-M3 platforms. The tool suite is an extension of the FELICS benchmarking framework and provides a new AEAD-specific low-level API that allows users to collect very fine-grained and detailed results for execution time, RAM consumption, and binary code size in a highly automated fashion.

2.1 Introduction

An Authenticated Encryption (AE) algorithm can be loosely defined as a symmetric cryptographic algorithm that is capable to (simultaneously) assure the confidentiality *and* authenticity of data [BR00; KY01]. A special form of AE, known as *Authenticated Encryption with Associated Data (AEAD)*, allows a part of the data to remain unencrypted, while still all data gets authenticated. The notion of AEAD was first formalized by Rogaway [Rog02] in 2002 and has applications in such areas as network packet encryption where the header (which contains the destination address) needs to be readable by routers, but should nonetheless be authenticated and integrity-protected. An AEAD algorithm takes a quadruple of the form (M, A, K, N) as input and outputs a tuple (C, T), where M is the message to be encrypted and authenticated, A is the associated data that gets authenticated only (but not encrypted), K is the secret key, N is a nonce, C is the ciphertext, and T is an authentication tag. Conversely, the decryption uses (C, A, K, N, T) as input and outputs the original message M if T is valid, or an error symbol $\perp$ otherwise. The two essential security goals an AEAD algorithm has to achieve are confidentiality and authenticity; a mathematically rigorous definition of both was given by Rogaway [Rog02]. Informally, confidentiality means that a passive adversary with access to C and T should not be able to deduce any information about M, except of its length. Authenticity generally refers to the ability to thwart forgery attacks, which means an active adversary should have a very low success probability when attempting to fabricate a (C, T)-tuple that the decrypting party will verify as authentic.

Initially, AEAD schemes were created by combining a block cipher in some mode of operation with a Message Authentication Code (MAC) algorithm. A clear disadvantage of this approach is the necessity of having two different primitives and requiring two passes over the message. Modern constructions including most submissions to NIST's lightweight cryptography standardization project, use a different approach, where a single algorithm is able to deliver authenticated encryption, with a single pass over the message.

In recent years, the cryptographic community has shown great interest in AEAD because of the CAESAR competition and the NIST call for lightweight primitives. CAESAR (short for Competition for Authenticated Encryption: Security, Applicability, and Robustness) is an already finished competition whose objective was to select a portfolio of AEAD algorithms. It followed the spirit of previous cryptographic competitions, such as the one that yielded the now omnipresent block cipher AES. In 2018, NIST officially announced the initiation of a process to solicit, evaluate, and standardize lightweight cryptographic algorithms—namely AEAD schemes and hash functions—that are suitable for constrained environments where the current standards can not provide acceptable performance. The motivation behind this initiative is the emergence of more and more application domains where constrained devices are interconnected to form the so-called Internet of Things (IoT). Security and privacy are extremely important in the IoT, but cannot always be provided by the currently standardized cryptosystems. This is because the severe constraints under which present (and future) IoT devices are expected to operate were not anticipated 20-25 years ago when many of the current NIST standards (e.g. AES, SHA-2) were designed.

2.1.1 Motivation and Research Needs

In response to NIST's call for proposals for lightweight AEAD algorithms and hash functions, a total of 57 candidates were submitted by March 29, 2019[1]. These candidates are currently evaluated in an open process taking various criteria into account, which include besides security (i.e. resistance against known cryptanalytic attacks) also practical aspects like performance and resource requirements (e.g. silicon area, memory footprint, code size) when implemented in hardware and software [Nat18]. NIST anticipates an initial (i.e. first-round) evaluation period of about six months to filter out candidates with obvious weaknesses and narrow the candidate pool for a more careful study and analysis in a second round. In total, the NIST estimates a duration of two to four years until the publication of a first draft standard and emphasizes that "the success of the lightweight crypto standardization process relies on the efforts of the researchers from the cryptographic community that provide security, implementation, and performance analysis of the candidates" [2].

Most papers introducing a new AEAD algorithm report some kind of results of some kind of performance evaluation on some kind of platform using some kind of implementation. Unfortunately, these results are usually not suitable for a comparison of the efficiency of two or more algorithms since it is not easily possible to take differences in the characteristics of the target platforms or differences in the simulation/measurement conditions into account. There is a need for a way to compare performance figures for many algorithms consistently and fairly so that designers and implementers of IoT applications can make better decisions regarding which algorithm is the most suitable one under a given set of efficiency requirements and resource constraints.

In the course of the CAESAR competition, the eBACS framework [BL09] was used for the bechmarking of the submitted AEAD algorithms. However, the original eBACS tools only support 64-bit Intel/AMD processors and high-end ARM models, mostly from the Cortex-A series, whereas many IoT devices are equipped with low-end microcontrollers, e.g. 8-bit AVR ATmega, 16-bit TI MSP430, or 32-bit ARM Cortex-M. These microcontrollers are optimized for small silicon area and low power consumption, which means they have totally different characteristics than their 64-bit counterparts. These differences manifest not only in the word size, but also the instruction set, the size of the register file, the latency of individual instructions, the degree of instruction-level parallelism, and many other aspects. For example, 64-bit Intel or ARM processors have a register space of 128 bytes (or even more when taking vector registers into account), whereas the MSP430 platform (which lies at the opposite end of the spectrum) provides 24 bytes altogether. Furthermore, most 8 and 16-bit microcontrollers can only execute shifts or rotations at a rate of one bit per cycle, whereas more powerful processors are capable to perform n-bit shifts/rotations in a single cycle. For all these reasons, benchmarking results generated with eBACS are of little use when it comes to the evaluation of AEAD algorithms on microcontrollers.

2.1.2 Aims and Contributions of this work

The present paper addresses the research needs identified above and puts forward a proposal for the benchmarking of lightweight AEAD algorithms. Our proposal aims

[1] Current at the time of the publication of the original article. Of the 57 candidates, 56 were selected for Round 1, 32 for round 2, and currently there are ten finalist candidates.

[2] See `https://csrc.nist.gov/projects/lightweight-cryptography/round-1-candidates` (accessed 2022-04-27).

to answer two basic questions that generally arise in the context of software benchmarking of cryptographic algorithms. The first question relates to the Application Program Interface (API) that implementations of a candidate algorithms have to follow to ensure a fair and consistent evaluation. We will argue in Section 2.2.2 that, for the purpose of benchmarking, it makes sense to use a low-level API sense ince it allows one to obtain more fine-grained results compared to a high-level API consisting of just the functions `encrypt` and `decrypt`. Furthermore, we introduce an API containing seven low-level functions, which we consider well suited for the benchmarking of AEAD algorithms. The second issue concerns the question of how to measure the execution time and other metrics of interest, which includes aspects like the length of the message M and the length of the associated data A. More concretely, how should the length-ratio of M and A be to get meaningful results? We will try to answer these questions in Section 2.2.1 through the definition of so-called evaluation scenarios that aim to mimic security-related operations commonly carried out by "real" IoT devices. More concretely, these scenarios are inspired by the need for AEAD operations in two networking protocols with relevance for the IoT, namely IEEE 802.15.4 (the most common PHY/MAC-layer protocol for low-rate wireless networks) and IPv6.

We implemented both the low-level API for AEAD and the evaluation scenarios in the form of an extension to the well-known and widely-used FELICS (Fair Evaluation of Lightweight Cryptographic Systems) framework [Cry16]. FELICS was originally created to support the collection benchmarking results for (lightweight) block ciphers on three embedded platforms: 8-bit AVR, 16-bit MSP430, and 32-bit ARM Cortex-M3. The full source code of FELICS is available under GPLv3 to increase the transparency and reproducibility of benchmarking results. Besides execution time, FELICS is also capable to determine the binary size and RAM footprint on the three currently supported platforms. The framework is modular, built on well documented and free compilers and tools, which allows easy extension of functionality and integration of new microcontroller platforms and evaluation scenarios. We tested the extended FELICS toolsuite using optimized C implementations of five AEAD algorithms (namely AES-GCM, ACORN, ASCON, Ketje-Jr, and NORX) that adhere to our low-level API. These tests confirm that FELICS-AEAD works properly and is able to collect large amounts of benchmarking results in an efficient and highly-automated fashion. An analysis of the collected benchmarking results for these five algorithms allows us to draw some conclusions about how basic design decisions like the organization of the "state" (i.e. whether the state is processed at a granularity of 32-bit words or 64-bit words) affect the performance on small microcontrollers.

2.2 FELICS Framework and its AEAD Extension

FELICS – ***F****air* ***E****valuation of* ***Li****ghtweight* ***C****ryptographic* ***S****ystems* – is a free and open source framework that assesses the efficiency of C and assembly implementations of lightweight cryptographic primitives on embedded devices. Following a modular design philosophy, the framework can easily be extended to accommodate new metrics, evaluation scenarios, and devices. FELICS is the core of an effort to increase the transparency in the in the performance analysis of lightweight algorithms and aims to facilitate a fair comparison of a large number of candidates.

2.2.1 Overview of Modules

FELICS is written in C, but also includes Bash and Python scripts. The framework was designed to work on Linux and allows the benchmarking of C and assembly implementations of cryptographic primitives that follow a set of pre-defined requirements. C was chosen because of its continuing popularity in the IoT and the fact that most reference implementations are written in this language. Furthermore, C code is highly portable, which is an important asset since there is no single dominating platform in the IoT. However, FELICS also supports the benchmarking of platform specific Assembler implementations to eliminate the impact of the compiler's ability (or inability) for code optimization. Hand-crafted Assembler code can take architecture-specific optimizations into account and has the potential to significantly outperform compiled C code.

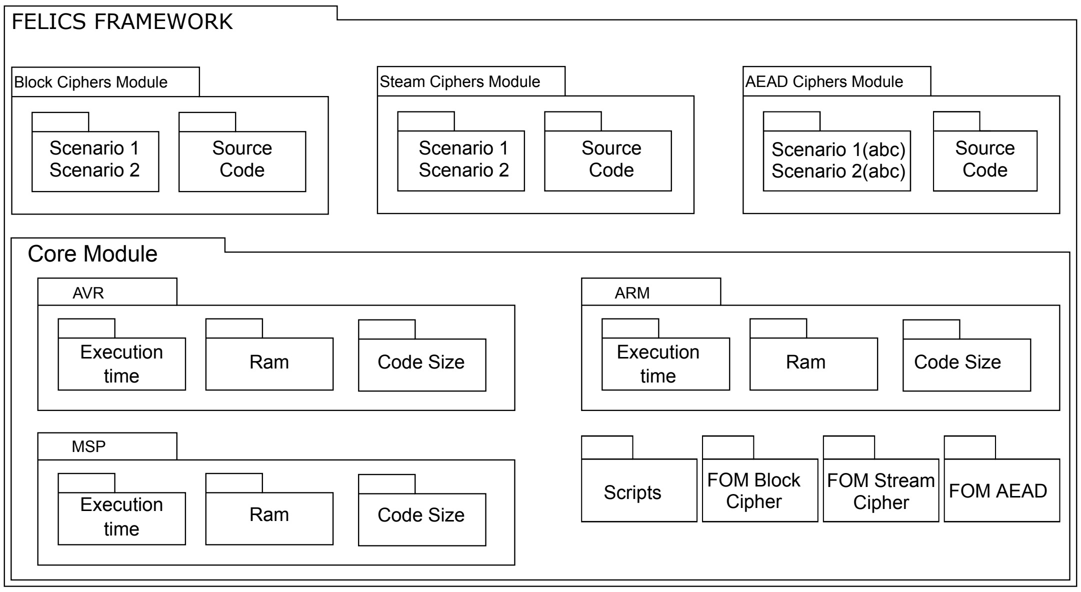

FIGURE 2.1: Modular structure of the FELICS benchmarking framework.

Core Module

The Core module, as the name implies, is the main part of the framework, and provides the tools necessary to collect the metrics for each of the supported devices. This module aims to facilitate the integration of new target devices and new metrics. Collection of metrics can be done individually or in batch mode. Beyond metrics collection, the Core also defines modules to debug and evaluate ciphers in a PC, mainly to aid in the implementation and integration process of new ciphers by the framework's users. A Python script for processing the generated CSV files and to assemble a ranking of candidates based on a so-called Figure-Of-Merit (FOM) is also present (see [Din+15a] for details).

The Core module is also responsible for exporting the extracted results. The framework main reporting format is CSV, but it also offers support to other output formats: Formatted raw text table, XML compatible with Microsoft Office Excel and LibreOffice Calc, MediaWiki table and Latex Table. Due to the modular nature of the framework, new formats can be added should the need for those arise.

Authenticated Encryption Module

This module allows the evaluation of lightweight AEAD ciphers. To allow the framework to extract the metrics, each cipher's implementation must follow the defined API.

A template for implementation, as well as implementations of identity ciphers, are provided with the module and can be used as a guide to help new users to integrate new implementations. The complete rules and step-by-step integration guide for cipher implementations can be found in the `README` file in the example cipher.

The framework supports cipher evaluation based on scenarios. Scenarios implement common real-world use cases, with practical relevance for IoT, with the main objective of generating realistic benchmark results that are meaningful in the real world. The current scenarios in the AEAD module of FELICS are divided into three main groups:

- **Debug and verification Scenario:** Also called Scenario 0, is mainly used for debugging purposes. It operates over a single block of input and allows the implementers to check their implementations on known test vectors.
- **IEEE 802.15.4 Scenarios:** These scenarios are based on the security needs of data communication in wireless sensor networks and other IoT applications using the IEEE 802.15.4 MAC/PHY-layer protocol. The maximum frame size of IEEE 802.15.4 is 127 bytes; the length of the header depends on various factors, such as the format of the source and destination addresses, but can not exceed 25 bytes. This leaves (at least) 102 bytes as frame payload. IEEE 802.15.4 supports three kinds of security services, namely (i) "Encryption Only" with AES in counter mode, (ii) "Authentication Only" with AES-CBC-MAC producing a MAC of either 32, 64, or 128 bits, and (iii) "Authenticated Encryption" using AES-CCM with the same MAC lengths.
  - **Scenario 1a:** Encryption of 102 bytes of data.
  - **Scenario 1b:** Authentication of 86 bytes of payload and 25 bytes of header. This scenario assumes that 16 bytes of payload are reserved to write the authentication tag.
  - **Scenario 1c:** Authenticated encryption of 86 bytes of payload and 25 bytes of header (which is authenticated but not encrypted). As with Scenario 1b, the authentication tag has a length of 16 bytes.
- **IPv6 Scenarios:** These scenarios are based on the use cases of IPv6 frames, as defined in RFC 2460. The MTU of IPv6 is at least 1280 bytes and the header has a fixed length of 40 bytes. Based on experiments with the Network Simulator NS-3, we found that the following message and associated data lengths serve as good representatives for real-world scenarios.
  - **Scenario 2a:** Encryption of 1240 bytes of data.
  - **Scenario 2b:** Authentication of 1224 bytes of payload and 40 bytes of header.
  - **Scenario 2c:** Authenticated encryption of 1224 bytes of payload and 40 bytes of header.

The IEEE 802.15.4 and IPv6 scenarios differ not only in the amount of data to be protected (127 bytes vs 1280 bytes), but also in the relation of data-length of AD-length. In the former case, the AD/D ratio is 0.29, whereas in the latter case the AD-length is negligible in relation to the D-length.

2.2.2 API for Authenticated Encryption

The FELICS API aims to offer a generic and well-specified interface for the most common operations performed by an AEAD algorithm. Different from other frameworks, the FELICS API is composed of seven low-level functions. While this may introduce difficulties for certain implementation techniques (e.g. bitslicing), the low-level API gives the framework more flexibility and allows one to obtain more fine-grained benchmarking results. Such fine-grained results can be useful, for example, when one wants to analyze *why* a given AEAD algorithm is more or less efficient and its competitors. Our seven functions are described below and their prototypes are given in Listing 2.1.

- **`Initialize`:** This function receives as parameters pointers to the algorithm's state, key, and nonce, and should execute the cipher's initialization procedures.
- **`ProcessAssocData`:** This function receives as parameters a pointer to the state, a byte stream of associated data, as well as its length.
- **`ProcessPlaintext`:** This function receives as parameters a pointer to the state, a byte stream of data, as well as the length of plaintext and ciphertext. The ciphertext should overwrite the plaintext.
- **`ProcessCiphertext`:** This function receives as parameters a pointer to the state, a byte stream of data, as well as the length of plaintext and ciphertext. The plaintext should overwrite the ciphertext.
- **`Finalize`:** This function receives as parameters pointers to the state and key, and executes the finalization steps on the internal state, preparing it for the authentication tag generation.
- **`GenerateTag`:** This functions receives as parameters a pointer to the internal state and the authentication tag and should write the appropriate information on the authentication tag.
- **`VerifyTag`:** This function received two pointers to authentication tags, and compares both. Returns `(int)(1)` if the tags match, and `(int)(0)` otherwise.

NIST specified a high-level API consisting of two functions (namely `aead_encrypt` and `aead_decrypt`), which submitters of AEAD candidates had to follow when they developed the (mandatory) reference implementation and an (optional) optimized implementation. While such a high-level API is convenient for software developers using AEAD algorithms, it is not necessarily a good choice for collecting benchmarking results, especially in Scenario 0. This is probably best explained taking the block-cipher benchmarks from [Din+15b] as example. Similar to AEAD, one can benchmark block ciphers using either a high-level or a low-level API. The former consists of generic functions for encrypting/decrypting of an arbitrary amount of data using a specified mode operation. On the other hand, the low-level API consists of two functions for each encryption and decryption, one to encrypt/decrypt a single block, and one to perform the encrytion/decryption key schedule. In order to minimize the overall development effort, the high-level functions can simply be implemented as wrappers over the low-level functions. However, using the low-level API for benchmarking

Listing 2.1: Function prototypes of the low-level AEAD API.

```
void Initialize(uint8_t *state, const uint8_t *key, const uint8_t *nonce);
void ProcessAssocData(uint8_t *state, uint8_t *assocData, size_t assocDataLen);
void ProcessPlaintext(uint8_t *state, uint8_t *message, size_t messageLen);
void ProcessCiphertext(uint8_t *state, uint8_t *message,size_t messageLen);
void Finalize(uint8_t *state, uint8_t *key);
void GenerateTag(uint8_t *state, uint8_t *tag);
int  VerifyTag(uint8_t *state, uint8_t *tag);
```

in Scenario 0 makes certain properties of ciphers more apparent than the high-level API. For example, RC5 is extremely fast, but has a very costly key schedule, which becomes immediately evident with benchmarking results obtained with the low-level API. Therefore, RC5 is unattractive for scenarios where the the amount of data to be encrypted or decrypted is small. This information is not so directly obvious when benchmarking results are generated with the high-level API.

2.2.3 Target Devices and Evaluation Metrics

For this framework, three widely used microcontrollers were chosen as representatives of the most used 8, 16, and 32-bit platforms used in the IoT. These microcontrollers have been optimized for small area and low power consumption. Their main characteristics are summarized in Table 2.1 and a brief description of each will follow on the next paragraphs.

Characteristic	AVR	MSP	ARM
CPU	8-bit RISC	16-bit RISC	32-bit RISC
Frequency	16 MHz	8 MHz	84 MHz
Registers	32	16	21
Architecture	Harvard	Von Neumann	Havard
Flash	128 KB	48 KB	512 KB
SRAM	4 KB	10 KB	96 KB
Supply voltage	4.6-5.5 V	1.8-3.6 V	1.6-3.6 V

Table 2.1: Key characteristics of the target microcontrollers.

The **AVR ATMega 128** is a microcontroller manufactured by Atmel, featuring 32 8-bit registers (`R0 - R31`) with single clock access time. Six of those registers can also be used as 16-bit indirect address pointers for data space. The instructions are executed within a two-stage, single-level pipeline, with most of its 133 instructions requiring a single cycle to execute. AVR processors are based on a modified Harvard architecture, where program and data are stored in separate physical memory regions in different physical addresses. Regarding memory, the ATmega128 comes with 128 KB Flash amd 4 KB SRAM.

The **MSP430F1611** microcontroller is a RISC CPU produced by Texas Instruments. It follows a Von Neumann architecture, and features 16 registers, with 12 being general purpose. Operations over registers take one clock cycle, while the other instructions depend on its format and addressing mode used. Memory wise, the MSP430 has one shared address space for special function registers, peripherals, RAM and FLASH. It has 48 KB of Flash and 10 KB of SRAM. Typical applications include medical devices and smart meters.

The 32-bit **Atmel SAM3X8 Cortex M3** is a RISC CPU that executes the Thumb-2 instruction set. This processor has a three-level pipeline and 13 general-purpose registers. It features 512 KB of Flash and 96 KB of SRAM divided into two banks of 64 KB and 32 KB. The Cortex-M3 is specially designed to achieve high performance in power-sensitive embedded applications, such as microcontrollers, automotive and industrial controllers, wireless networking, and others. This processor runs at a maximum frequency of 84 MHz.

For cipher evaluation, three metrics are used: Execution time, RAM usage, and code size. These metrics were chosen because they outline the main characteristics of the implementations. Secondary metrics, such as energy consumption were not included mainly due to being closely related to the basic metrics.

Execution time consists in measuring w number of cycles necessary to execute a given operation. This metric is extracted by using either a cycle-accurate simulator a development board. Extraction of cycle-counter uses AVRORA [TLP05] for the AVR processor, and `MSPDebug` [Bee15] for MSP. Extraction of cycle counter on ARM is done via the automatic insertion of code to read ARM's system time registers. One important detail regarding ARM's measurements is that there may exist variations in the extracted numbers, due to different instructions being generated at compilation time and memory alignment of test data, as well as different manufacturer's implementation of the ISA.

RAM consumption is a combination of stack and data requirements. The stack consumption describes the maximum amount of RAM used to store local variables and return addresses after interruptions and system calls. The data requirement represents the static RAM usage and is given by the size of the constants stored in the device's RAM. Static RAM consumption is measured using the GNU `size` tool. The stack consumption is measured using a `gdb` client and the target simulator or development board. At the beginning of each operation to be measured, the stack is filled with a memory pattern. At the end of the function execution, the memory is compared with the initial pattern, and the number of modified bytes gives the stack consumption.

Alternatively, stack requirements could be calculated statically via a call graph, using the generated assembly instructions. This method, though, is not able to solve recursive functions, neither calls to functions in the standard C lib. Using a `gdb` client with a well-tested simulator is less error-prone than building such an analysis tool from scratch and was the preferred method in the framework.

Code size is measured in bytes and quantifies the amount of storage an operation or evaluation scenario occupies in the non-volatile memory of the target device. It is measured using the GNU `size` tool on the appropriate object files. To obtain the overall code size, the framework simply sums the size of the `text` and `data` sections, which contain, respectively, the executable instructions generated by the compiler and the static variables that are initialized with a non-zero value. The `bss` section is not taken into account since none of the benchmarked operations and scenarios use uninitialized static variables.

Figure of Merit

Due to space limitations, normally only a subset of data can be correctly shown in publications. To aid in the classifications of the evaluated ciphers, FELICS introduces the *Figure-of-Merit*(FOM), that can be used to rank the analyzed ciphers. For each implementation i and platform d, a performance indicator p_{i_d} that aggregates the metrics from $M = \{$ execution time, RAM consumption, code size $\}$ as

$$p_{i,d} = \sum_{m \in M} w_m \frac{v_{i,d,m}}{\min_i(v_{i,d,m})}$$

where $v_{i,d,m}$ is the value of the metric m for the implementation i on the platform p; and w_m is the relative weight for the metric m, with $w_m = 1$ by default for all platforms. Then, for each cipher and the selected set of best implementations i_{AVR}, i_{MSP}, and i_{ARM} (one for each platform) the FOM is calculated as the average performance indicator across the three platforms:

$$\text{FOM}(i_{AVR}, i_{MSP}, i_{ARM}) = \frac{p_{i_{AVR}} + p_{i_{MSP}} + p_{i_{ARM}}}{3}$$

2.3 Analyzed AEAD Algorithms

In this section, we briefly describe the ciphers implemented in FELICS, as an example and initial work for the framework. These ciphers were chosen for their relevance in the context of IoT and lightweight cryptography, as well for being part of an ongoing effort of standardizing AEAD schemes.

ACORN

Acorn is an AEAD scheme created by Hongjun Wu, and finalist of the CAESAR competition. It features a stream-cipher-like construction based on six concatenated linear feedback shift registers. The cipher's design benefits lightweight hardware implementations since the processing can be done in a bitwise fashion. It also allows the parallel computation of 32 steps of the cipher, which is beneficial for both hardware and software implementations. Another characteristic of ACORN, it does not need to check message length and does not need to pad messages to a multiple of the block size. ACORN has advantages over AES-GCM, namely being more efficient in terms of hardware resources and energy consumption, and in constrained devices (without AES instructions and polynomial computing circuits) ACORN is more efficient than AES in software [Wu16].

AES-GCM

The Galois/Counter mode is a mode of operation for 128-bit block ciphers, widely used together with the AES block cipher for its efficiency and performance. GCM is used in MACSec Ethernet Security, IEEE 802.11ad wireless protocols, Fibre Channel security protocols, and is also included in the NSA Suite B Cryptography, as well as various other software [MV04]. AES-GCM gains great performance results by using pipelined or parallelized operations, as well as using specific hardware instructions, such as Intel's `PCLMULQDQ` and AES-NI instructions. These characteristics, although, are a target of criticism, since parallel processing is not suited for embedded and limited devices, which thus results in a reduction of performance in those devices.

ASCON

ASCON is a family of AEAD ciphers, finalist of the CAESAR competition. It was designed by Christoph Dobraunig et al. in 2014. The main goal of ASCON is to achieve a very low memory footprint, both in hardware and software implementations, and

Cipher	Block	Key	Nonce	State	Tag
NORX	384	128	128	512	128
ACORN	1	128	128	293	128
Ketje-Jr	16	128	48	200	128
ASCON	64	128	128	320	128
AES-GCM	128	128	96	1824	128

TABLE 2.2: Parameters of the evaluated ciphers, in bits.

still provide an adequate combination of security, speed, and size, with focus on the last. ASCON is based on the Sponge Design, being similar to SpongeWrap and MonkeyDuplex constructions [Dob+14]. The permutation of ASCON is an SPN designed to provide low-cost fast diffusion. The SBox used is an improved version of the χ mapping of Keccak, and the LBox uses a function similar to SHA-2's Σ functions. ASCON was not designed to compete with very fast parallel AEAD schemes running on unconstrained devices, but on the other hand, it features good instruction parallelism, which makes it achieve good characteristics on constrained devices.

Ketje

Ketje is a family of four AEAD algorithms, aimed to memory-constrained devices and that strongly relies on nonce uniqueness for security. It was designed by Guido Bertoni et al. and is a third-round candidate of the CAESAR competition. Ketje is based on a reduced round version of Keccak, over a MonkeyDuplex and MonkeyWrap constructions [Ber+b]. One interesting characteristic of Ketje is the support for sessions: Without the need for a key and a new nonce, communicating parties can exchange metadata-plaintext pairs, with the generated tag authenticating the complete exchange of messages since the begging of the communication process. Of the four algorithms in the Ketje family, two are aimed towards compact implementations –Ketje Jr and Sr– and two are research ciphers aimed towards high speed –Ketje Minor and Major.

NORX

NORX is a family of AEAD ciphers created by Jean-Philippe Aumasson et al. in 2014. NORX supports associated data both as header and trailer. The algorithm also supports arbitrary parallelism in the payload processing step and is optimized for hardware and software implementations, with a specially SIMD friendly construction. NORX is based on ChaCha's permutation, with the integer addition replaced by an ARX approximation, which –according to the designers– allows simplified cryptanalysis and improves hardware efficiency [AJN14].

2.4 Preliminary Results

Using the FELICS extension for authenticated encryption described in Section 2, we benchmarked optimized C implementations of the five AEAD algorithms on three platforms and for two evaluation scenarios plus Scenario 0, which is mainly for debugging and verification. Table 2.2 summarizes the main characteristics of the specific variants of the AEAD algorithms we implemented.

The FELICS framework allows ranking all these implementations according to their

Cipher		AVR			MSP			ARM			FOM
		Size	Mem	Time	Size	Mem	Time	Size	Mem	Time	
NORX	S1a	4702	214	135640	3992	214	66738	1474	214	17227	4.3
	S1b	3936	223	90728	3482	223	53035	1148	223	10089	4.0
	S1c	5028	207	124062	4216	207	75727	1634	207	16685	4.5
ASCON	S1a	3734	190	519420	5656	190	599643	1712	190	80316	9.4
	S1b	3734	199	340671	5656	199	395564	1712	199	52958	8.9
	S1c	3734	183	534908	5656	183	619523	1712	183	83118	9.4
Ketje-Jr	S1a	5156	165	290446	6248	165	346867	3564	165	138867	9.4
	S1b	5156	174	211749	6248	174	254923	3564	174	99490	9.8
	S1c	5156	158	311949	6248	158	372720	3564	158	148381	9.7
ACORN	S1a	3292	191	337818	3170	191	456972	1954	191	191869	10.0
	S1b	3292	200	408914	3170	200	551501	1954	200	236235	15.7
	S1c	3292	184	464381	3170	184	626192	1954	184	267168	12.5
AES-GCM	S1a	6578	374	889573	6798	374	2137251	6096	374	1086449	41.5
	S1b	5944	383	447505	6782	383	1150450	6028	383	565606	34.0
	S1c	6578	367	975184	6798	367	2369572	6096	367	1197073	44.6

Table 2.3: Results for Scenario 1 (IEEE 802.15.4). For each platform and each cipher, the best implementation results are reported in case more than one implementation was available in the framework. The code size and memory consumption are specified for the whole scenario (and not just the AEAD algorithm alone), which includes the 127-byte IEEE 802.15.4 frame to be encrypted and/or authenticated. The smaller the Figure-of-merit, the better is the implementation of a cipher.

execution time, RAM footprint, or code size in any scenario on any platform. Table 2.3 summarizes the results of Scenario 1, which is inspired by the need for security in the IEEE 802.15.4 protocol. This scenario actually consists of three sub-scenarios with different operations and slightly different lengths of the data to be encrypted and/or authenticated. However, all three sub-scenarios have in common that the amount of data is relatively small, namely between 86 and 102 bytes, due to the 127-byte MTU –maximum transmission unit– of the IEEE 802.15.4 protocol. If associated data is processed, its length is roughly one fourth of the data-length. Concretely, in Sub-scenario 1a ("encryption only"), 102 bytes of data are encrypted, whereas in Sub-scenario 1b ("authentication only") the size of the data is 86 bytes and the size of the associated data is 25 bytes. Finally, in Scenario 1c ("authenticated encryption") 86 bytes of data are encrypted and $86 + 25 = 111$ bytes are authenticated. NORX is the clear winner in all three sub-scenarios, followed by ASCON and Ketje-Jr, which perform very similar in all three sub-scenarios. However, the FOM score of the latter two algorithms is more than twice higher than that of NORX.

Finally, Table 2.4 shows the results of Scenario 2, which deals with security for the IPv6 protocol. This scenario is again split into three sub-scenarios, similar to the sub-scenarios in the context of IEEE 802.15.4 described above. However, the amount of data to be encrypted is much larger, around 1200 bytes, while the amount of associated data is relatively small; more concretely, the ratio between data and associated data is roughly 30:1. Again, NORX is the clear winner in all three sub-scenarios, followed by ASCON and Ketje-Jr. However, compared to the IEEE 802.15.4 scenarios,

Cipher		AVR			MSP			ARM			FOM
		Size	Mem	Time	Size	Mem	Time	Size	Mem	Time	
NORX	S2a	4702	1376	800313	3992	1376	501290	1474	1376	109933	4.1
	S2b	3936	1376	424601	3482	1376	246263	1148	1376	46113	3.7
	S2c	5028	1376	814467	4216	1376	508728	1634	1376	111361	4.2
ASCON	S2a	3292	1353	1811457	3170	1353	2454962	1954	1353	1013715	8.5
	S2b	3292	1353	1136110	3170	1353	1541295	1954	1353	644411	10.5
	S2c	3292	1353	1916720	3170	1353	2595469	1954	1353	1077068	8.7
Ketje-Jr	S2a	5156	1327	3026956	6248	1327	3623707	3564	1327	1481660	12.6
	S2b	5156	1327	1527941	6248	1327	1860262	3564	1327	751536	13.3
	S2c	5156	1327	3007966	6248	1327	3601416	3564	1327	1471405	12.5
ACORN	S2a	3734	1352	6174633	5656	1352	7109127	1712	1352	947367	13.9
	S2b	3734	1352	3146041	5656	1352	3619665	1712	1352	479574	14.2
	S2c	3734	1352	6112583	5656	1352	7039689	1712	1352	938358	13.6
AES-GCM	S2a	6578	1536	9807655	6798	1536	23748153	6096	1536	12036393	64.4
	S2b	5944	1536	3526008	6782	1536	9531538	6028	1536	4564667	54.2
	S2c	6578	1536	9812008	6798	1536	23796554	6096	1536	12050336	63.6

Table 2.4: Results for Scenario 2 (IPv6). For each platform and each cipher, the best implementation results are reported in case more than one implementation was available in the framework. The code size and memory consumption are specified for the whole scenario (and not just the AEAD algorithm alone), which includes the 1280-byte IPv6 packet to be encrypted and/or authenticated. The smaller the Figure-of-merit, the better is the implementation of a cipher.

the difference between ASCON and Ketje-Jr is much bigger. Similar to before, the FOM score of NORX is significantly better than that of the runner-up ASCON.

It is interesting to observe that NORX is in both scenarios speed-wise much better than the other candidates. NORX outperforms its CAESAR competitors by a factor of at least two; in some extreme cases, NORX is even five times faster than the second-best algorithm. This significant difference begs for more analysis and raises the question of what design decisions make an AEAD algorithm efficient (or inefficient) on small microcontroller platforms. However, this question is difficult to answer since the efficiency of AEAD designs depends on many different factors, some of which are architecture-independent, i.e. affect the performance on 8, 16, 32, and 64-bit platforms similarly, whereas others are architecture-dependent in the sense that they impact the performance across platforms differently. An example of the latter is the organization of the state, i.e. whether the state is processed at a granularity of 32-bit words or 64-bit words. The benchmarked version of NORX processes the state in 32-bit words, whereas ASCON, ACORN, and Ketje-Jr operate on 64-bit words. Organizing the state in 64-bit quantities is the natural choice for designs aiming at high performance on Intel/AMD and 64-bit ARM processors as it allows one to exploit the full word-size of these processors, but may lead to suboptimal performance on smaller microcontroller platforms, which is due to three reasons.

First, C compilers for 8-bit AVR and 16-bit MSP microcontrollers (e.g. `mspgcc`) are, in general, not very good at handling 64-bit words (i.e. operands of type `uint64_t`). We assume this is because outside cryptography there are very few application domains where a programmer really needs a 64-bit integer on an 8 or 16-bit microcontroller.

NORX128 uses 32-bit words, which seems to make it much easier for a C compiler to generate efficient code than for the other CAESAR candidates that process 64-bit words. The second reason is the small register space of 8 and 16-bit microcontrollers. For example, the MSP430 architecture comes with only twelve 16-bit general-purpose registers, which means it would theoretically be possible to hold three 64-bit words in the register file. However, in practice, this is not the case since always one or two registers are needed for temporary results and often also one register has to be set to 0. Therefore, it can be expected that no more than two 64-bit words can be kept in registers at any time, but it may be possible to accommodate five 32-bit words when the cipher's state is organized in 32-bit words. Finally, the third reason why 64-bit words can entail suboptimal performance is ARM-specific and relates to the fact that one of the two operands of an arithmetic/logical instruction is fed through a barrel-shifter before it enters the ALU, which means shifts and rotations can be executed "for free" together with other instructions. However, on a 32-bit ARM microcontroller, shifts and rotations are only free for 32-bit operands, but not for 64-bit quantities.

2.5 Comparison with other Benchmarking Tools

Besides FELICS, there exist a few other tools for the benchmarking of cryptographic algorithms, of which eBACS and XXBX are the most closely related ones. eBACS (short for ECRYPT Benchmarking of Cryptographic Systems) was developed during the ECRYPT II project to evaluate the performance of cryptographic algorithms on Intel/AMD processors and high-end ARM models capable to run Linux (e.g. the Cortex-A series). It features modules for measuring the performance of public-key cryptosystems (called eBATS), stream ciphers (eBASC), hash functions (eBASH), and authenticated encryption algorithms (eBAEAD). Those modules operate all under a common framework called SUPERCOP (System for Unified Performance Evaluation Related to Cryptographic Operations and Primitives) that allows benchmarking of C, C++ and assembly implementations. It comes with a large collection of implementations of cryptographic algorithms and automatically compiles source code using different compilers and compiler options. The execution time is extracted via a cycle counter (accessed through assembler code) for many different lengths of input data. Since execution time is the only metric measured by this framework, implementations are optimized solely for speed.

The eXternal Benchmarking eXtension [WBG10] is an extension for the SUPERCOP framework developed with the objective of benchmarking hash functions on different microcontrollers in the context of the SHA-3 competition. XBX was the first project to measure, in a unified manner, the performance of cryptographic primitives built for different devices using the same evaluation methodology. In support for the now finished CAESAR competition, XBX was extended for AEAD algorithms and the ability to measure power consumption. However, apart from a 1-page summary of this so-called XXBX extension [Car+18] (published in 2017), we are not of aware any further papers describing concrete details of its inner working, which indicates that XXBX is still under development.

Low-Level API.

eBACS (and also XXBX) require AEAD implementations to follow a simple high-level API consisting of just two basic functions, namely `aead_encrypt` and `aead_decrypt`.

This simplicity ensures that the API is easy to use (and hard to misuse), even for inexperienced software developers, but yields very coarse-grained results when applied to benchmarking. FELICS-AEAD, on the other hand, defines a low-level API comprising the seven functions specified in Listing 2.1. This low-level API offers a high degree of flexibility and allows for easy implementation of different kinds of security services, including the high-level functions of eBACS, for which nothing more than simple wrappers are needed. Consequently, adhering to the low-level API does not introduce more development effort than the high-level functions of eBACS. However, the low-level API enables a more fine-grained evaluation of AEAD algorithms since not only their overall execution times can be compared but also the times needed for initialization, encrypting/decrypting the data, processing the associated data, and generating/verifying the authentication tag. All these timings are valuable for algorithm designers when trying to analyze *why* a given AEAD algorithm is faster or slower than others. The fine-grained benchmarking results obtained with the low-level API may also be useful when one has to find the most suitable AEAD algorithm (out of a pool of candidates) for the encryption and/or authentication of a certain amount of data and associated data, respectively.

Evaluation Scenarios.

eBACS measures the execution time of AEAD algorithms for combinations of data lengths and associated data lengths ranging from 0 to 2048 bytes in steps of one byte. These more than four million combinations have to be multiplied by the number of compiler options (i.e. optimization levels), which makes the collection of benchmarking results extremely computation-intensive and costly, especially when a large number of AEAD implementations have to be evaluated. The target platforms of eBACS (Intel/AMD and high-end ARM processors) are powerful enough to execute such a workload in an acceptable time, but this is not the case for resource-constrained 8 and 16-bit microcontrollers that can only be accessed via a debug probe and have to be programmed separately for each implementation. Using cycle-accurate instruction-set simulators is also not a solution since most of them lack a stable way of scripting to automate the verification of test vectors and the recording of cycle counts. These issues were the main reason to introduce the two evaluation scenarios (and six sub-scenarios) described in Section 2.2.1. Namely, by defining very specific use cases that resemble real-world security services in the IoT, FELICS-AEAD becomes capable to evaluate a large number of implementations in a reasonable amount of time. The two scenarios are intended to have very different characteristics and requirements for AEAD algorithms. For example, the amount of data in Scenario 1 is relatively small and the length of the associated data is roughly a quarter of the data length. On the other hand, the amount of data in Scenario 2 is much higher, but the associated data amounts to only a small fraction of the data-length.

Figure-of-Merit

eBACS measures only the execution time of AEAD implementations, which makes it relatively easy to rank candidates by e.g. comparing their average throughput in cycles/byte. In contrast, FELICS-AEAD determines not only the execution time but also the memory footprint and code size of an implementation on each of the three supported platforms. This is reasonable since both RAM and ROM (resp. flash) are usually scarce resources in the IoT. However, taking three different metrics for each AEAD implementation into account makes a comparison of the benchmarking results relatively difficult, which is why FELICS allows the user to define a Figure-of-Merit

(FOM) that combines execution time, RAM footprint, and code size into a single number. The FOM metric can use different weight factors for the three metrics, but by default, they have equal weight and, consequently, the execution time is considered to be equally important as RAM footprint and code size.

2.6 Conclusions and Remarks

In this paper, we introduced an extension to FELICS, a free and open-source benchmarking framework for the evaluation of AEAD algorithms. The main motivation behind this development is to give the designers of AEAD algorithms a fair, comprehensive and consistent way of evaluating their algorithms in the context of lightweight embedded devices, as well as a consistent way of comparing performance metrics between different algorithms. More specifically, this paper provided three contributions: (i) an API that allows a fine-grained evaluation of algorithms, while still maintaining design flexibility for the designers; (ii) a series of real-world based evaluations scenarios, allowing a fair comparison of algorithms based on their predicted future use; and (iii) preliminary results with a small set of well-known AEAD algorithms that demonstrate the framework's practical value. Thanks to its modular design, FELICS is very flexible and can be extended to support new metrics, new scenarios, and new devices. Furthermore, new implementations of AEAD algorithms can easily be added to the framework. With that in mind, we encourage the cryptographic community to contribute optimized C and Assembler implementations of AEAD candidates submitted to the NIST lightweight crypto project and support in this way the fair and transparent evaluation of AEAD algorithms.

Chapter 3

Multiplatform Benchmarking of AEAD

This chapter is an adaptation of the article titled "An Evaluation of the Multi-Platform Efficiency of Lightweight Cryptographic Permutations"[SG21]. The paper was accepted for publication on SECITC 2021: *14th International Conference on Security for Information Technology and Communications.*

Permutation-based symmetric cryptography has become increasingly popular over the past ten years, especially in the lightweight domain. More than half of the 32 second-round candidates of NIST's lightweight cryptography standardization project are permutation-based designs or can be instantiated with a permutation. The performance of a permutation-based construction depends, among other aspects, on the *rate* (i.e. the number of bytes processed per call of the permutation function) and the execution time of the permutation. In this paper we analyze the execution time and code size of assembler implementations of the permutation of ASCON, GIMLI, SCHWAEMM, and XOODYAK on an 8-bit AVR and a 32-bit ARM Cortex-M3 microcontrollers. Our aim is to ascertain how well these four permutations perform on microcontrollers with very different architectural and micro-architectural characteristics such as the available register capacity or the latency of multi-bit shifts and rotations. We also determine the impact of flash wait states on the execution time of the permutations on Cortex-M3 development boards with 0, 2, and 5 wait states. Our results show that the throughput (in terms of permutation time divided by rate when the capacity is fixed to 256 bits) of the permutation of ASCON, SCHWAEMM, and XOODYAK is similar on ARM Cortex-M3 and lies in the range of 41.1 to 48.4 cycles per rate-byte. However, on an 8-bit AVR ATmega128, the permutation of SCHWAEMM outperforms its counterparts of ASCON and XOODYAK by a factor of 1.59 and 1.48, respectively.

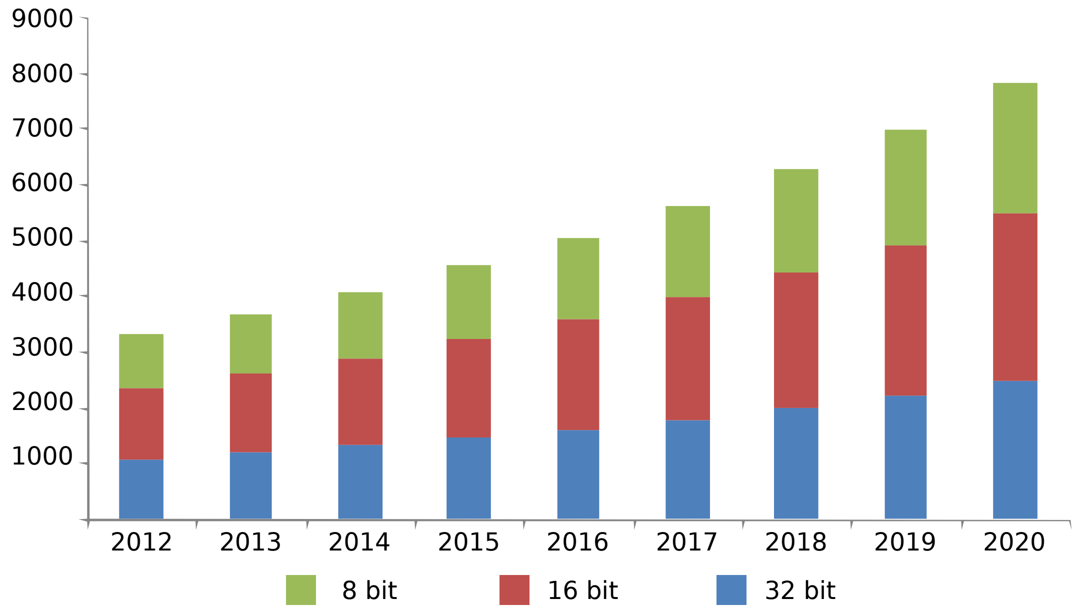

FIGURE 3.1: North American microcontroller market by product (8-bit, 16-bit, 32-bit) in million units (source: Radiant Insights Inc. [Rad15])

3.1 Introduction

The term *Internet of Things (IoT)* describes the evolution of the Internet from a computer network to a network that connects various kinds of smart devices and enables them to communicate with each other or transmit data to central servers. This development started roughly 15 years ago, when more and more "everyday objects," ranging from household appliances over business machines to vehicles, became equipped with microcontrollers and transceivers for wireless communication (e.g. ZigBee, Bluetooth, WiFi). These devices differ greatly in terms of computing power, but also regarding their data transmission speeds and run-time memory capacities. At one end of the spectrum are e.g. modern cars, which are equipped with powerful processors, while e.g. battery-operated miniature sensor nodes at the opposite end of the spectrum commonly feature only a small 8-bit or 16-bit microcontroller. Already today, approximately twice as many "smart things" are connected to the Internet than ordinary computers like PCs or laptops, and this proportion will grow rapidly over the next couple of years [Tel17]. Internet-enabled smart devices can be found in basically all areas of our life, from home automation over industrial production ("Industry 4.0") to transportation and logistics.

The IoT can be seen as a large ecosystem populated by highly diverse and heterogeneous devices, which come in all shapes and sizes. Therefore, it is little surprising that there exist dozens of different (and largely incompatible) microcontroller platforms, operating systems, and wireless communication standards for the IoT, many of which are optimized to serve a certain application domain with specific requirements and constraints. This heterogeneity of IoT devices is in stark contrast to the "monoculture" in the realm of classical computers like PCs or laptops, where the 64-bit Intel architecture has a market share of well over 90%. Nonetheless, 64-bit Intel processors represent only a small fraction of the IoT altogether, which is (quantitatively) dominated by microcontrollers with rather modest computational capabilities. Figure 3.1 shows a forecast of the development of the North American microcontroller market until 2020, split up in 8-bit, 16-bit, and 32-bit architectures [Rad15]. The North American market was estimated to be over 3700 million units in 2013 and is expected to reach some 8000 million units in 2020, i.e. the compound annual growth rate is more than 11.2% in the period from 2014 to 2020. 32-bit microcontrollers constitute the fastest growing product segment over the forecast period, driven mainly by an increased demand for higher processing capabilities and the expected reduction in unit prices. Currently, the ARM architecture is the undisputed leader in the 32-bit segment, but it faces fierce competition by ESP32 and RISC-V. There is also a growing demand for 16-bit microcontrollers (e.g. MSP430, 68HC16) due to the need for high level of precision in embedded processing and the development of intelligent and real-time functions in the automotive domain [Rad15]. The 8-bit platforms (e.g. AVR, PIC) are expected to retain their market share and continue to be widely used for automotive and industrial applications [Mor20].

Since there is no single dominating microcontroller platform in the IoT, it is essential that a cryptographic algorithm delivers consistently high performance on a wide variety of 8, 16, and 32-bit architectures. This is far from trivial to achieve since, for example, a 32-bit ARM Cortex-M3 microcontroller has significantly different architectural and micro-architectural characteristics than an 8-bit AVR ATmega microcontroller. The Cortex-M3 has 16 registers, of which 14 are available for general use, i.e. the general-purpose register space amounts to 448 bits. AVR microcontrollers, on the other hand, have 32 general-purpose registers, but each of them can only store eight bits of data, yielding a usable register space of 256 bits. ARM and AVR also differ greatly in their ability to execute multi-bit shifts or rotations, which are performance-critical operations of various symmetric cryptosystems. The arithmetic/logic unit of a Cortex-M3 comes with a fast barrel shifter capable to shift or rotate a 32-bit word by an arbitrary number of bits in a single cycle. Furthermore, a shift or rotation can be combined with most data-processing instructions, in which case they become practically "free" [Arm21]. More specifically, the second operand of most arithmetic or logical instructions can be shifted or rotated (before the actual operation is executed) without increasing the instruction latency. However, the situation is much different for 8 and 16-bit architectures, as most of them have only single-bit shift and rotate instructions, which means that shifting a register by n bits requires (at least) n clock cycles. This can make multi-bit shifts and rotations very costly, especially when the length of the operand to be shifted or rotated exceeds the capacity of a single register. For example, rotating a 32-bit word on an 8-bit AVR microcontroller (stored in four registers) can, depending on the rotation amount, require more than 20 clock cycles.

A *cryptographic permutation* is a bijective mapping within $\mathbb{Z}_2^b$, designed to behave as a random permutation, i.e. a permutation chosen randomly from the set of all possible permutations that operate on b bits. The width b of a cryptographic permutation is usually between 200 (for cryptosystems targeting the lightweight domain) and 1600 [Ber+12]. Permutation-based cryptography emerged approximately 15 years ago as a sub-area of research in the field of symmetric cryptography and started to attract particular interest when the hash function KECCAK [Ber+11e] and the stream cipher SALSA [Ber08] became popular[1]. Permutations are extremely flexible and versatile primitives, similar to block ciphers, and can be used to construct e.g. hash functions, message authentication codes, pseudo-random bit-sequence generators, stream

[1]In October 2012, the U.S. National Institute of Standards and Technology (NIST) selected KECCAK as winner of the SHA-3 hash competition [Nat15]. Roughly 1.5 years later, in April 2014, Google announced that a TLS cipher suite using CHACHA20 (a variant of SALSA) for symmetric encryption

ciphers, and authenticated encryption algorithms [Ber+11b; Ber+12]. However, unlike a block cipher, a permutation does not have a key schedule and needs to be efficient only in one direction since the inverse permutation is normally never used. Past research in the area of permutation-based cryptography can be split into two main categories; the first is about the design (and security analysis) of permutation-based constructions and "modes of use" built on top of them, while the second category is concerned with the permutations themselves. Representative work in the former category includes besides the classical sponge [Ber+11b] and duplex [Ber+11d] construction also various kinds of constuctions/modes that aim to boost performance through a higher bitrate (e.g. full-state absorption [MRV15], BEETLE mode [Cha+18]) or via a parallel application of a sponge or a permutation (e.g. KANGAROOTWELVE [Ber+18], FARFALLE [Ber+17c]), as well as modes with "built-in" countermeasures against certain physical attacks (e.g. ISAP [Dob+20]). Research in the second category deals mainly with the design of permutations and their efficient (and side-channel resistant) implementation in hardware and/or software. The majority of the published permutations are either classical Addition-Rotation-XOR (ARX) designs, e.g. SALSA [Ber08], or can be classified as "AndRX" variants, e.g. KECCAK-f [Ber+11e], NORX F^l [AJN16].

Permutation-based cryptography is well suited for resource-limited devices (e.g. RFID tags, wireless sensor nodes, smart cards), which is evidenced by the fact that roughly half of the 32 second-round candidates of NIST's currently-ongoing standardization effort for lightweight cryptography use a permutation as underlying primitive [Nat21]. However, despite a broad body of research in the area of permutation-based cryptography, surprisingly little is known about the performance of state-of-the-art permutations on small microcontrollers. There exist, of course, a lot of benchmarking results for the second-round candidates of NIST's lightweight crypto project[2], but these benchmarks specify only the execution time of the full authenticated encryption (resp. hash) algorithms and not that of the permutation alone. These timings are relatively poor indicators for the efficiency of the underlying permutation since they also include various "auxiliary" operations. For example, designs based on the BEETLE mode, such as the NIST candidate SCHWAEMM [Bei+20c], include a feedback function ρ through which data is injected into (and extracted from) the state. Furthermore, some optimized implementations of permutations that operate on 64-bit words, like KECCAK-f[1600] and ASCON's p [Dob+21], adopt the bit-interleaving method [Ber+11e] to speed up rotations on 32-bit ARM processors. This bit-interleaving makes the injection/extraction of data to/from the state more costly, whereby the actual penalty factor depends on how fast the permutation itself is. The benchmarks for full authenticated encryption or hash algorithms do not even allow one to reason about the *relative* efficiency of their permutations due to differences in the bit-rates. Unfortunately, the lack of detailed implementation results makes the design of new permutations a challenging task since it is not easily possible to compare the execution time and code size with the state-of-the-art.

In this paper, we analyze and compare the multi-platform (resp. cross-platform) efficiency of four cryptographic permutations that are part of candidates of the current lightweight cryptography standardization project of the National Institute of Standards and Technology (NIST) [Nat21]. These four candidates are ASCON [Dob+21],

will be their default option to secure HTTPS connections on devices without AES hardware acceleration [Bur14].

[2]See `https://github.com/usnistgov/Lightweight-Cryptography-Benchmarking` (accessed 2020-10-10).

Gimli [Ber+17a], Schwaemm [Bei+20c], and Xoodyak [Dae+20], all of which come with algorithms for Authenticated Encryption with Associated Data (AEAD) and hashing. In addition, they have in common that the permutation width is very similar (i.e. between 320 and 384 bits) and they all consist of only simple arithmetic/logic operations (Schwaemm is a classical ARX construction, while the other three can be classified as "AndRX" designs, i.e. they use the logical AND operation or OR operation as a source of non-linearity). We evaluate the execution time and code size of these four permutations with highly-optimized Assembler implementations for ARM Cortex-M3 and AVR ATmega128 microcontrollers, whereby we applied the same general optimization strategies and invested a similar amount of optimization effort for each implementation so as to ensure a fair evaluation. By focusing solely on the permutations, we aim to make their relative performance more transparent and generate new insights to their multi-platform efficiency, which are not immediately apparent when one compares the execution times collected by other benchmarking initiatives. We also assess how basic design decisions, e.g. shift/rotation amounts, impact the performance of the permutations on 32-bit ARM and 8-bit AVR platforms.

3.2 Overview of the Permutations

In this section, we briefly review the main properties of the four permutations we consider in this paper, which are the permutations of the NIST candidates Ascon, Gimli, Schwaemm, and Xoodyak. Except for Gimli, they all made it to the final round of the evaluation process [Nat21]. Gimli was eliminated in the second round, but we still include it in our study since its permutation is well known and has inspired a number of other designs.

ASCON

Ascon is not only one of the 10 finalists of NIST's standardization project in lightweight cryptography, but was also selected for the final portfolio of the CAESAR competition. The main AEAD instance of the Ascon suite is Ascon-128 and offers 128-bit security according to [Dob+21]. It is based on the so- called Monkey Duplex mode [Ber+12] with a stronger keyed initialization and keyed finalization function, respectively, which means the underlying permutation is carried out with an increased number of rounds. Said permutation operates on a 320-bit state (organized in five 64-bit words) by iteratively applying a round function p. The number of rounds is $a = 12$ in the initialization and finalization phase, and $b = 6$ otherwise; the corresponding permutations are referred to as p^a and p^b in the specification. Ascon-128 processes associated data as well as plaintext/ciphertext with a rate of $r = 64$ bits, i.e. the capacity is 256 bits. The hash function of the Ascon suite is a classical sponge construction.

Ascon's round function p is SPN-based and comprises three parts: (i) the addition of an 8-bit round constant c_r to a 64-bit state-word, (ii) a substitution layer that operates across the five words of the state and implements an affine equivalent of the S-box in the χ mapping of Keccak, and (iii) a permutation layer consisting of linear functions that are similar to the Σ functions in SHA2 and performed on each state-word individually. The S-box maps five input bits to five output bits and is applied to each column of the state, whereby the five state-words are arranged upon each other. It is normally implemented in a bit- sliced fashion using logical ANDs and XORs. The permutation layer performs an operation of the form $x = x \oplus (x \ggg n_1) \oplus (x \ggg n_2)$

on each word x of the state with $n_1 \in \{1, 7, 10, 19, 61\}$ and $n_2 \in \{6, 17, 28, 39, 41\}$ [Dob+21].

Gimli

The second-round NIST candidate Gimli consists of the AEAD algorithm Gimli-Cipher and the hash function Gimli-Hash. Both are claimed to provide 128 bits of security against all known attacks, and Gimli-Cipher even uses a 256-bit key to “reduce concerns about multi-target attacks and quantum attacks” [Ber+17a]. The underlying 384-bit permutation is called Gimli-24 and was presented at CHES 2017. Gimli-Cipher is a conventional duplex construction with a capacity of 256 bits, i.e. the rate is 128 bits. On the other hand, Gimli-Hash is an ordinary sponge and also uses a rate of 128 bits. Unfortunately, the permutation has weak diffusion, which makes it possible to build a full-round distinguisher of relatively low complexity [Fló+21]. Though this distinguisher on the permutation does not immediately threaten the security of Gimli-Cipher and Gimli-Hash, the NIST decided to not promote Gimli to the final round.

The Gimli-24 permutation was designed to reach high performance across a broad range of platforms, from high-end 64-bit CPUs with vector extensions to small 8-bit microcontrollers, as well as FPGAs and ASICs. Its 384-bit state is represented as a 3×4 matrix of 32-bit words. Each of the 24 rounds consists of three operations: (i) a non-linear layer in the form of a 96-bit SP-box that is applied to each column of the matrix, (ii) a linear mixing layer in every second round, and (iii) a constant addition in every fourth round. The SP-box itself is inspired by Norx and can be efficiently implemented using logical operations (32-bit AND, OR, and XOR), left shifts by 1, 2 and 3 bits, as well as rotations by 9 and 24 bits. On the other hand, the linear layer performs swap operations on row 0 of the matrix: a small-swap every fourth round (starting from round 1), and a big-swap also every fourth round (starting from round 3).

Sparkle

The Sparkle suite submitted to NIST consists of four instances of the AEAD algorithm Schwaemm, targeting security levels of 128, 192, and 256 bits, as well as two instances of the hash function Esch with digest lengths of 256 and 384 bits. All instances are built on top of the Sparkle permutation family, which consists of three members that differ by the width (i.e. the state size) and the number of steps they execute. Schwaemm is based on the highly-efficient Beetle mode of use [Cha+18], whereas Esch can be classified as a sponge construction. The main instance of Schwaemm uses the 384-bit variant of the Sparkle permutation, i.e. Sparkle384, with a rate of 256 bits. This variant is also used for Esch256, the main instance of the hash function Esch. Besides Sparkle384, there exists also a smaller and a larger version of the permutation with a width of 256 and 512 bits, respectively (see [Bei+20c] for details).

Sparkle384 is a classical ARX design, optimized for high speed on a wide range of 8, 16, and 32-bit microcontrollers. The permutation is performed with a big number of steps, namely 11, for initialization, finalization, and separation between the processing of associated data and the secret message, while a slim (i.e. 7-step) version is used to update the intermediate state. From a high-level point of view, the permutation has an SPN structure and comprises three main parts: (i) a non-linear layer consisting of six parallel ARX-boxes, (ii) a simple linear diffusion layer, (iii) the addition of a step counter and round constant to the 384-bit state. The ARX-box is called ALZETTE and can be seen as a small 64-bit block cipher that operates on two 32-bit words and performs additions modulo 2^{32}, logical XORs, and rotations by 16, 17, 24, and 31 bits [Bei+20c]. On the other hand, the linear layer is, in essence, a Feistel round with a linear Feistel function, followed by a swap of the left and right half of the state.

Xoodoo

XOODYAK is a highly versatile cryptographic scheme that is suitable for a wide range of symmetric-key functions including hashing, pseudo-random bit generation, authentication, encryption, and authenticated encryption. At its heart is XOODOO, a lightweight 384-bit permutation [Dae+18]. The XOODYAK suite submitted to the NIST lightweight crypto project includes an AEAD algorithm and a hash function; both are built on the Cyclist mode of operation [Dae+20]. To perform authenticated encryption, Cyclist has to be initialized in keyed mode with a 128-bit key and nonce, respectively, after which associated data can be absorbed at a rate of 352 bits (i.e. 44 bytes), whereas plaintext/ciphertext gets processed at a rate of 192 bits. On the other hand, when Cyclist is operated in hash mode, the rate is 128 bits (i.e. 256 bits of capacity).

XOODOO is inspired by both KECCAK and GIMLI in the sense that the state has the same size and is represented in the same way as in GIMLI, though the round function is similar to KECCAK [Ber+11e]. Consequently, the state has the form of a 3×4 matrix of 32-bit words, which can be visualized via three horizontal 128-bit planes (one above the other), each consisting of four 32-bit lanes. It is also possible to view the 384-bit state as 128 columns of three bits lying upon another (i.e. each bit belongs to a different plane). The XOODOO permutation executes 12 iterations of a round function of five steps: a column-parity mixing layer θ, a non-linear layer χ, two plane-shifting layers (ρ_{west} and ρ_{east}) between them, and a round-constant addition. Both ρ layers move bits horizontally and perform lane-wise rotations of planes as well as rotations of lanes by 11, 1, and 8 bits to the left. On the other hand, in the parity-computation part of θ and in the χ layer, state-bits interact only vertically, i.e. within 3-bit columns. The θ layer mainly executes XORs and left-rotations by 5 and 14 bits. Finally, the non-linear layer χ applies a 3-bit S-box to each column of the state, which can be computed using logical ANDs, XORs, and bitwise complements.

3.3 Implementation and Evaluation

To ensure a fair and consistent evaluation of the four permutations, we applied the same implementation and optimization strategy to each permutation, and we put a similar effort into optimizing each implementation. This section gives a brief overview of our optimization strategy for ARM and AVR, and describes how we optimized and benchmarked the permutations. In total, we evaluated eight implementations (four for ARM and also four for AVR), half of which we developed from scratch, namely the ARM implementation of SPARKLE384 and the AVR implementations of ASCON, SPARKLE384, and XOODOO, whereas the remaining four are based on Assembler source code provided by the designers (with minor modifications to ensure a fair and consistent evaluation).

3.3.1 Optimization Strategies

The assembly implementations for the ARM Cortex-M3 platform [Arm16] are purely speed-optimized, which means whenever there was a trade-off to be made between execution time and code size, we opted for the optimization that led to the best performance. This implies, for example, that the main loop of each permutation is fully unrolled to eliminate the loop overhead. Round constants are not kept in tables in flash or RAM, but put into registers on the fly using `movw` and `movt` instructions or, if they are less than 12 bits long, directly encoded into an instruction as intermediate value. Such speed-optimized implementations have been developed by the designers of Ascon, Gimli, and Xoodoo; we used these implementations as starting point and made some small modifications to increase the readability of the source code (e.g. by using macros) and ensured that they all adhere to the specification of the ARM Application Binary Interface (ABI). For example, the ABI specification requires that the stack pointer is double-word (i.e. 8 bytes) aligned at a public interface; when necessary we modified the source code to ensure full ABI compliance. We also translated the assembler source code of Gimli from the GNU assembler syntax to the syntax used by Keil MicroVision so that its execution time can be determined with Keil's cycle-accurate simulator and by execution on development boards using the GNU toolchain for ARM. The original ARM implementation of Ascon provided by its designers was written in the form of "inlined" assembly code for the permutation. We converted this implementation into a separate assembly function to ensure consistency across all permutations. The fourth permutation, which is Sparkle384, was implemented by us from scratch.

Our assembly implementations of the permutations for the 8-bit AVR architecture [Mic20] aim for small (binary) code size instead of high speed. Therefore, we refrained from code-size increasing optimization techniques like (full) loop unrolling as otherwise the code size would become unreasonably large. This can be exemplified using the AVR assembler implementations of Gimli (provided by its designers) as case study. One of these implementations is size-optimized and, therefore, relatively small (less than 800 bytes), whereas the other is speed-optimized (with fully unrolled loop) and has a code size of more than 19 kB. For comparison, the code size of the fully-unrolled ARM implementation is less than 4 kB. However, it has to be taken into account that flash capacity for storing program code is, in general, more scarce on small and cheap devices with an 8-bit microcontroller than on devices equipped with a more powerful 32-bit ARM microcontroller. We developed the assembly implementations of Ascon, Sparkle384, and Xoodoo from scratch since, at the time we started with our evaluation of the permutations, no optimized AVR assembler code existed for them. However, we took over the size-optimized AVR implementation of the Gimli permutation developed by its designers since it complies with our optimization strategy. We put a similar effort into optimizing the AVR implementation of the permutations to ensure a fair and consistent evaluation.

3.3.2 Benchmarking.

We evaluated the execution time of both the AVR and the ARM implementations through simulation with a cycle-accurate instruction set simulator, namely the simulator contained in Atmel Studio 7 and Keil MicroVision 5.24, respectively. Execution times obtained by simulation with Atmel Studio are, in general, very close to the timings on "real" hardware. Unfortunately, this is usually not the case for simulation results for ARM since, as mentioned on the Keil website[3], the simulator assumes ideal conditions for memory accesses and "does not simulate wait states for data or code fetches." Therefore, the timings obtained with the simulator should be seen as lower bounds of the actual execution times one will get on a real Cortex-M3 device. In order to get realistic performance figures, we also measured the execution time of the permutations on three development boards with a different number of flash wait states. The first board is the STM32 VL Discovery, which is equipped with an STM32F100RBT6B Cortex-M3 microcontroller clocked with a nominal frequency of 24 MHz. Due to this relatively low clock frequency, the microcontroller can access flash memory without wait states. Our second board is again an STM32 board, but a more sophisticated one, namely the STM32 Nucleo-64. It comes with an STM32F103RBT6 Cortex-M3 microcontroller clocked with a frequency of 72 MHz. At this frequency, flash accesses require two wait states. Finally, the third board is an Arduino Due [Ard12], which houses an Atmel SAM3X8E Cortex-M3 microcontroller clocked with a frequency of 84 MHz. When operated with its standard configuration, flash accesses require 5 wait states. However, the performance impact of this high number of wait states is, to some extent, mitigated by a "flash accelerator."

3.4 Results

Table 3.1 compares the execution time and code size of speed-optimized (i.e. fully unrolled) assembly implementations of the four permutations Ascon, Gimli, Sparkle384, and Xoodoo. These execution times have been determined via simulation with the cycle-accurate instruction set simulator of Keil MicroVision 5.24 using a generic Cortex-M3 model as target device. The times range from 387 clock cycles (for Ascon) to 1041 clock cycles (Gimli). However, when comparing symmetric cryptographic primitives, the throughput (in cycles per byte) is often more meaningful than the raw execution time. For example, when comparing block ciphers, the throughput in terms of execution time divided by block size allows one to take into account that different algorithms can have different block sizes. Similarly, when comparing permutations, one can obtain throughput figures by dividing the computation time by either the width of the permutation or the rate of the associated AEAD algorithm. In the case of our four permutations, the corresponding AEAD rates are eight bytes (Ascon-128), 16 bytes (Gimli-Cipher), 24 bytes (Xoodyak), as well as 32 bytes (Schwaemm256-128). Unfortunately, when using the rate of the associated AEAD algorithm to determine the throughput, the evaluation takes into account the efficiency of the permutation *and* the efficiency of the mode of the AEAD algorithm. However, since we want to evaluate and compare the efficiency of the permutation alone, we decided to calculate the throughput under the assumption that all four permutations are used to instantiate one and the same mode of use (namely a classical sponge) with one and the same capacity (namely 256 bits, which corresponds to 128 bits of security). This means Ascon has a rate of eight bytes, and the three other permutations a rate of 16 bytes.

The last column of Table 3.1 specifies the throughput (in cycles per byte) of the permutations calculated in this way, i.e. by dividing the execution time by the rate under the assumption that the permutation is used to instantiate a classical sponge with a capacity of 256 bits. Xoodoo reaches the best throughput since it needs only 41 clock cycles per rate-byte, closely followed by Ascon and Sparkle384, which

[3]See `https://www2.keil.com/mdk5/simulation` (accessed 2020-09-14)

Permutation	Code size (bytes)	Exec. time (clock cycles)	Throughput (cc/rate-byte)
ASCON-128 (6 rounds)	1364	387	48.38
Gimli (24 rounds)	3950	1041	65.06
SPARKLE384 (7 steps)	2820	781	48.81
Xoodoo (12 rounds)	2376	657	41.06

TABLE 3.1: Code size, execution time, and throughput of speed-optimized ARMv7-M assembly implementations of the four permutations on a Cortex-M3 microcontroller.

have almost identical throughput. The throughput of GIMLI is clearly the worst of all four permutations and about 1.5 times lower than that of XOODOO. In terms of code size, ASCON is the clear winner, while GIMLI has the largest code size and is more than twice as big as ASCON.

Permutation	Code size (bytes)	Exec. time (clock cycles)	Throughput (cc/rate-byte)
ASCON-128 (6 rounds)	888	6434	804.25
Gimli (24 rounds)	778	23699	1481.19
SPARKLE384 (7 steps)	866	8068	504.25
Xoodoo (12 rounds)	906	11972	748.25

TABLE 3.2: Code size, execution time, and throughput of size-optimized AVR assembly implementations of the four permutations on an ATmega128 microcontroller.

Table 3.2 contains the code size, execution time and throughput (in terms of permutation time divided by the rate, assuming a capacity of 256 bits) of code-size optimized AVR assembly implementation of the four permutations on an 8-bit ATmega128 microcontroller [Mic11]. The execution times were simulated using the cycle-accurate instruction set simulator that is part of Atmel Studio 7. Apparently, the obtained AVR timings are significantly worse (by at least an order of magnitude) than the execution times of the permutations on ARM. This massive performance penalty can be explained by the different optimization goals (i.e. size vs. speed) and, more importantly, by the completely different characteristics of the architectures as mentioned in Section 1 (e.g. register space, latency of multi-bit shifts and rotations). In terms of throughput, SPARKLE384 is now the clear winner, followed by ASCON and XOODOO. In contrast to ARM, ASCON slightly outperforms XOODOO on AVR. While on ARM the top-3 permutations were throughput-wise relatively close to each other, we see a significant difference on AVR since the throughput of ASCON is 1.60 times worse than the throughput of SPARKLE384, and the throughput of XOODOO is about 1.62 times worse. We emphasize again that these results are based on size-optimized implementations, which means all four permutations can reach better throughput rates when optimized for speed. Such speed-optimized implementations were developed in the course of Rhys Weatherley's benchmarking project for AVR and are available online[4]. Interestingly, these benchmarking results indicate that the *relative* performance of the corresponding AEAD algorithms is very similar to our throughput results for the permutations, namely SCHWAEMM256-128 is significantly faster than ASCON-128 and XOODYAK.

[4]See `https://rweather.github.io/lightweight-crypto/performance_avr.html` (accessed 2020-09-14).

Permutation	Keil μVision (simulation)	VL Discovery 0 wait states	Nucleo-64 2 wait states	Arduino Due 5 wait states
ASCON-128 (6 rounds)	387	389	601 (1.54)	472 (1.21)
Gimli (24 rounds)	1041	1043	1656 (1.59)	1287 (1.23)
SPARKLE384 (7 steps)	781	782	1196 (1.53)	936 (1.20)
Xoodoo (12 rounds)	657	659	1014 (1.54)	795 (1.21)

Table 3.3: Execution time of the four permutations as determined by simulation with Keil MicroVision using a generic Cortex-M3 model and measurement on Cortex-M3 development boards with 0, 2, and 5 flash wait states (values in parentheses are the performance penalties over the execution time on the VL Discovery board, which has 0 flash wait states).

As mentioned in the previous section, the simulation results obtained with Keil MicroVision may differ from the execution time on "real" Cortex-M3 hardware since the Keil simulator does not take flash wait states into account. The purpose of flash wait states is to compensate the difference of the maximum clock frequency with which the microcontroller core and the flash memory can be clocked. Modern Cortex-M3 microcontrollers can reach clock frequencies of more than 200 MHz, which is far above the maximum frequency of flash memory (usually between 16 to 32 MHz). Therefore, we decided to assess the impact of flash wait states on the performance of the four permutations by measuring their execution time on the three Cortex-M3 development boards mentioned in the previous section, namely an STM32 VL Discovery (no flash wait states), an STM32 Nucleo-64 (2 flash wait states), and an Arduino Due (5 flash wait states). However, the SAM3X8E microcontroller on the Arduino board contains a "flash accelerator," which is essentially a small buffer located between the microcontroller core and the flash memory, to mitigate the impact of the wait states. Table 3.3 shows the measured execution times of the four permutations on these boards. The timings on the VL Discovery are almost identical to those obtained through simulation with Keil MicroVision, which confirms that the Keil simulator is indeed cycle-accurate. On the other hand, the execution times on the Nucleo-64 board are significantly worse (by factors of between 1.53 and 1.60) than the results on the Discovery board and the timings reported by the simulator. These results also show that flash wait states do not impact each permutation to the same extent since the penalty factor for Ascon is higher than the penalty factor for Sparkle384. The timings on the Arduino Due are better than the timings on the Nucleo-64, despite the larger number of wait states, which is due to the afore-mentioned flash accelerator.

3.5 Conclusions

Since there is no single dominating platform in the IoT, designers of lightweight cryptographic algorithms have to aim for multi-platform efficiency, i.e. efficiency on a wide range of microcontroller architectures with highly diverse and divergent characteristics. In this paper we analyzed to what extent the permutations of the NIST candidates Ascon, Gimli, Sparkle and Xoodyak achieve this goal, whereby we used 32-bit ARM Cortex-M3 and 8-bit AVR as evaluation platforms. We benchmarked speed-optimized assembler implementations for ARM, using source code provided by the designers, and size-optimized assembler implementations for AVR, which we mainly developed from scratch. Our results show that the throughput (i.e. permutation time divided by rate when the capacity is fixed to 256 bits) of Ascon, Sparkle384, and Xoodoo is very similar on ARM and differs only by a few cycles per rate-byte. On the other hand, on 8-bit AVR, SPARKLE384 significantly outperforms ASCON and XOODOO by a factor of 1.63 and 1.65, respectively. One reason for this discrepancy between the ARM and AVR results is the cost of multi-bit shifts and rotations on the latter, which has a significant impact on the overall execution time. However, the shifts and rotations do not impact the four permutations in the same way; the impact on SPARKLE is relatively small since, as stated by the designers in [Bei+20c], the rotation amounts were specifically chosen in a way that facilitates high speed on AVR. On the other hand, the designers of ASCON and XOODOO either "over-optimized" their permutations for ARM, or they neglected efficiency on small 8 and 16-bit microcontrollers. On a more positive note, the results for SPARKLE demonstrate that it is possible to design a permutation for multi-platform efficiency.

Part III

Sparkle family of algorithms

Chapter 4

Sparkle family of algorithms

This chapter is based on the specification of the Sparkle family of of ciphers and hash functions, which was published in a special issue of TOSC, dedicated to second round candidates of the NIST lightweight standardization process[Bei+19]. Design minutiae and cryptanalitical proofs are not discussed in this dissertation, as my main contribution was towards the implementation aspects, and helping guide the design into a direction that yielded high performance code.

My main contribution in this chapter was in Implementation Aspects. During the design period of Alzette and their related hash and AEAD functions, my main work was guiding the design towards components that would yield good performance on our target architectures, e.g. usage of $8n \pm 1$ bit rotations, combinations of arithmetic/logic operations with shifts that can use ARM's barrel shifter, and sizing the components in a way to maximize register-bank usage and minimize load-store operations.

4.1 Introduction

With the advent of the Internet of Things (IoT), a myriad of devices are being connected to one another in order to exchange information. This information has to be secured. Symmetric cryptography can ensure that the data those devices share remains confidential, that it is properly authenticated and that it has not been tampered with.

As such objects have little computing power—and even less so that is dedicated to information security—the cost of the algorithms ensuring these properties has to be as low as possible. To answer this need, the NIST has called for the design of authenticated ciphers and hash functions providing a sufficient security level at as small an implementation cost as possible.

In this document, we present a suite of algorithms that answer this call. All our algorithms are built using the same core, namely the Sparkle family of permutations. The authenticated ciphers, Schwaemm, provide confidentiality of the plaintext as well as both integrity and authentication for the plaintext and for additional public associated data.

The hash functions, Esch, are (second) preimage and collision-resistant. Our aim for our algorithms is to use as few CPU cycles as possible to perform their task while retaining strong security guarantees and a small implementation size. This speed will allow devices to use much fewer CPU cycles than what is currently needed to ensure the protection of their data. To give one of many very concrete applications of this gain, the energy demanded by cryptography for a battery-powered microcontroller will be decreased.

In summary, our goal is to provide *fast software encryption* for all platforms.

4.2 Esch and Schwaemm: Hash and AEAD

Both are cryptographic algorithms that were designed to be lightweight in software (i.e., to have small code size and low RAM footprint) and still reach high performance on a wide range of 8, 16, and 32-bit microcontrollers. Section 4.6 gives an overview of software implementation options for different platforms. Esch and Schwaemm can also be well optimized to achieve small silicon area and low power consumption when implemented in hardware. Hardware implementation aspects (including a proposal for a lightweight hardware architecture for the Sparkle permutation) are discussed in Section 4.6.1.

Our schemes are built from well-understood principles, i.e., the sponge (resp. duplex-sponge) construction based on a cryptographic permutation, which, for example, the NIST hashing standard SHA-3 employs as well. Our underlying permutation, Sparkle, follows an ARX construction like SHA-2 and Chacha/Salsa but, unlike most ARX constructions, we provide security guarantees with regard to differential and linear cryptanalysis thanks to the *long trail strategy (LTS)*. The particular structure it imposes is also convenient to investigate other attacks (integral, impossible differential, etc.) and thus to argue about the security of our algorithms against

After the main design phase, I worked on the reference code for NIST's evaluation, and optimized implementations and benchmarking for the competition part of the LWC call.

Due to the nature of my contribution, design minutiae and cryptanalitical proofs are not discussed in this chapter.

them. The LTS is a strategy which was first used to design the lightweight block cipher `Sparx`, presented at ASIACRYPT 2016 [Din+]. Several independent research teams have already analyzed this algorithm and their results have bolstered our confidence in this design approach.

4.2.1 The Hash Function Esch

A *hash function* takes a message of arbitrary length and outputs a digest with a fixed length. It should provide the cryptographic security notions of *preimage resistance*, *second preimage resistance* and *collision resistance*. The main instance of ESCH (i.e., the primary member of the submission for the hash functionality) is ESCH256 which produces a 256-bit digest, offering a security level of 128 bits with regard to the above mentioned security goals. It is based on the permutation family SPARKLE384 (see Section 4.3). We also provide the member ESCH384 based on the permutation family SPARKLE512, which produces a 384-bit digest and offers a security level of 192 bits. Both of those hash functions serve as the basis for two Extendable-Output Functions (XOFs): XOESCH256 and XOESCH384.

The name ESCH stands for

Efficient, **S**ponge-based, and **C**heap **H**ashing.

It is also the part of the name of a small town in southern Luxembourg, which is close to the campus of the University of Luxembourg. ESCH is pronounced [ˈɛʃ].

4.2.2 The Authenticated Cipher Schwaemm

A scheme for *authenticated encryption with associated data (AEAD)* takes a key and a nonce of fixed length, as well as a message and associated data of arbitrary size. The encryption procedure outputs a ciphertext of the message as well as a fixed-size authentication tag. The decryption procedure takes the key, nonce, associated data and the ciphertext and tag as input and outputs the decrypted message if the tag is valid, otherwise a symbolic error $\perp$. An AEAD scheme should fulfill the security notions of *confidentiality* and *integrity*. Users *must not* reuse nonces for processing messages in a fixed-key instance.

The main instance of SCHWAEMM (i.e., the primary member of the submission for the AEAD functionality) is SCHWAEMM256-128 which takes a 256-bit nonce, a 128-bit key and outputs a 128-bit authentication tag. It achieves a security level of 120 bits with regard to confidentiality and integrity. We further provide three other instances, i.e., SCHWAEMM128-128, SCHWAEMM192-192, and SCHWAEMM256-256 which differ in the length of key, nonce and tag and in the achieved security level.

The name SCHWAEMM stands for

Sponge-based **C**ipher for **H**ardened but **W**eightless **A**uthenticated **E**ncryption on **M**any **M**icrocontrollers

It is also the Luxembourgish word for "*sponges*". SCHWAEMM is pronounced [ˈʃvɛm].

4.3 Underlying permutation: Sparkle

It is a family of cryptographic permutations based on an ARX design. Its name comes from the block cipher SPARX [Din+], which SPARKLE is closely related to. SPARKLE is basically a SPARX instance with a wider block size and a fixed key, hence its name:

SPARx, but **K**ey **LE**ss.

We provide three versions corresponding to three block sizes, i.e., SPARKLE256, SPARKLE384, and SPARKLE512. The number of steps used varies with the use case as our design approach is not *hermetic* (see Section 4.4.2).

4.4 Key Features

Both SCHWAEMM and ESCH employ the well-known sponge construction. The underlying SPARKLE family of permutations was designed from scratch, but based on well-known and widely accepted principles, to achieve high security *and* high efficiency. The following two subsections give an overview of the main features of SCHWAEMM and ESCH. A more detailed discussion is provided in Chapter 4.6.

4.4.1 What is their efficiency based on?

In the context of cryptographic software, the term *efficiency* is commonly associated with fast execution times, low run-time memory (i.e., RAM) requirements, and small code size. However, these three metrics are mutually exclusive since standard software optimization techniques to increase performance, such as loop unrolling or the use of look-up tables to speed up certain operations (e.g., SubBytes in AES), come at the expense of increased code size or increased RAM footprint or both. On the other hand, a cryptographic hardware implementation is called *efficient* when it achieves small silicon area and, depending on the requirements of the target application, low power consumption, low latency, or high throughput, whereby one of these metrics can be optimized at the expense of the other(s).

Small State size. Both SCHWAEMM and ESCH are characterized by a relatively small state size, which is only 256 bits for the most lightweight instance of SCHWAEMM described in this document (achieving a security level of 120 bits) and 384 bits for the lightest variant of ESCH. Having a small state is an important asset for lightweight cryptosystems for several reasons. First and foremost, the size of the state determines to a large extent the RAM consumption (in the case of software implementation) and the silicon area (when implemented in hardware) of a symmetric algorithm. In particular, software implementations for 8 and 16-bit microcontrollers with little register space (e.g., Atmel AVR or TI MSP430) can profit significantly from a small state size since it allows a large fraction of the state to reside in registers, which reduces the number of load and store operations. On 32-bit microcontrollers (e.g., ARM Cortex-M series) it is even possible to keep a full 256-bit state in registers, thereby eliminating almost all loads and stores. The ability to hold the whole state in registers does not only benefit execution time, but also provides some intrinsic protection against side-channel attacks [BDG16]. Finally, since SCHWAEMM and ESCH consist of very simple arithmetic/logical operations (which are cheap in hardware), the overall silicon area of a standard-cell implementation is primarily determined by storage required for the state.

Extremely lightweight permutation. The SPARKLE permutation is a classical ARX design and performs additions, rotations, and XOR operations on 32-bit words. Using a word-size of 32 bits enables high efficiency in software on 8, 16, and 32-bit platforms; smaller word-sizes (e.g., 16 bits) would compromise performance on 32-bit platforms, whereas 64-bit words are problematic for 8-bit microcontrollers. The rotation amounts (16, 17, 24, and 31 bits) have been carefully chosen to minimize the execution time and code size on microcontrollers that support only rotations by one bit at a time. An implementation of Sparkle for ARM microcontrollers can exploit their ability to combine an addition or XOR with a rotation into a single instruction with a latency of one clock cycle. On the other hand, a small-area hardware implementation can take advantage of the fact that only six arithmetic/logical operations need to be supported: 32-bit XOR, addition modulo 2^{32}, and rotations by 16, 17, 24, and 31 bits. A minimalist 32-bit Arithmetic/Logic Unit (ALU) for these six operations can be well optimized to achieve small silicon area and low power consumption.

Consistency across security levels. Schwaemm and Esch were designed to be consistent across security levels, which facilitates a parameterized software implementation of the algorithms and the underlying permutation Sparkle. All instances of Schwaemm and Esch can use a single implementation of Sparkle that is parameterized with respect to the block (i.e., state) size and the number of steps. Such a parameterized implementation reduces the software development effort significantly since only a single function for Sparkle needs to be implemented and tested.

Even higher speed through parallelism. The performance of Schwaemm and Esch on processor platforms with vector engines (e.g., ARM NEON, Intel SSE and AVX) can be significantly increased by taking advantage of the SIMD-level parallelism they provide, which is possible since all 32-bit words of the state perform the same operations in the same order. Hardware implementations can trade performance for silicon area by instantiating several 32-bit ALUs that work in parallel.

4.4.2 What Is Their Security Based On?

We have not traded security for efficiency. Our detailed security finds that our algorithms are safe from all attacks we are aware of with a comfortable security margin. Overall, the security levels our primitives provide are on par with those of modern symmetric algorithms but their cost is lower. Our hash functions are secure against preimage, second preimage and collision search. Our authenticated cipher provide confidentiality, integrity and authentication.

The Security of Sponges. The security of our schemes is based on the security of the underlying cryptographic permutations and the security of sponge-based modes, more precisely the sponge-based hashing mode and the Beetle mode for authenticated encryption (which is based on a duplexed sponge). The sponge-based approach has received a lot of attention as it the one used by the latest NIST-standardized hash function, SHA-3. We re-use this approach to leverage both its low memory footprint and the confidence cryptographers have gained for such components.

The Literature on Block Cipher Design. The design of the Sparkle family of permutations is based on the decades old SPN structure which allows us to decompose its analysis into two stages: first the study of its substitution layer, and, second, the study of its linear layer. The latter combines the Feistel structure, which has been used since the publication of the DES [Des], and a linear permutation with a high branching number, like a vast number of SPNs such as the AES [Aes]. To combine these two types of subcomponents, we rely on the design strategy that was used for the block cipher Sparx: the long trail strategy. Our substitution layer operates on 64-bit branches using ARX-based S-boxes, where ARX stands (modular) Addition, Rotation and XOR. The study of the differential and linear properties of modular addition in the context of block cipher can be traced back to the late 90's. The fact that the block size of the ARX component (the *ARX-box*, named Alzette[1] [Bei+20b]) is limited to 64 bits means that it is possible to investigate it thoroughly using computer assisted methods. The simplicity and particular shape of the linear layer then allows us to deduce the properties of the full permutation from those of the 64-bit ARX-box.

Components Tailored for Their Use Cases. When using a permutation in a mode of operation, two approaches are possible. We can use a "*hermetic*" approach (see [Ber+11c, Section 8.1.1]), meaning that no distinguishers are known to exist against the permutation. This security then carries over directly to the whole function (e.g. to the whole hash function or AEAD scheme). The downside in this case is that this hermetic strategy requires an expensive permutation which, in the context of lightweight cryptography, may be too much.

At the opposite, we can use a permutation which, on its own, cannot provide the properties needed. The security is then provided by the coupling of the permutation and the mode of operation in which it is used. For example, the winner of the CAESAR competition Ascon [Dob+16] and the third-round CAESAR candidate Ketje [Ber+16], both authenticated ciphers, use such an approach. The advantage in this case is a much higher efficiency as we need fewer rounds of the permutation. However, the security guarantees are *a priori* weaker in this case as it is harder to estimate the strength needed by the permutation. It is necessary to carefully assess the security of the specific permutation used with the knowledge of the mode of operation it is intended for.

For Sparkle (and thus for both Esch and Schwaemm), we use the latter approach: the permutation used has a number of rounds that may allow the existence of some distinguishers (in the sense that we do not claim that the permutation behaves like one would expect from a randomly-drawn permutation). However, using a novel application of the established long trail strategy, we are able to prove that our algorithms are safe with regard to the most important attack vectors (*differential attacks*, i.e., the method used to break SHA-1 [Ste+17], and *linear attacks*) with a comfortable security margin. We thus get the best of both worlds: we do not have the performance penalty of a hermetic approach but still obtain security guarantees similar to those of a hermetic design.

4.4.3 More Security Features

Security under Random Nonces. All instances of Schwaemm, with the exception of Schwaemm128-128, permit nonce sizes higher than 192 bits. Therefore, a collision in randomly chosen nonces is not expected to happen before 2^{92} encryptions are performed. Therefore, the security of the authenticated encryption schemes is not affected when the user employs them with nonces chosen uniformly at random for each encryption process.[2]

Integrity Security without Restrictions on the Number of Forgery Attempts. The Beetle mode of operation allows us to use a small internal state

[1]Alzette is pronounced [alzɛt].

[2]Since Schwaemm128-128 allows nonces of 128 bits, the same claim on the security under randomly chosen nonces holds when the number of encryptions is $\ll 2^{64}$.

together with a high rate to ensure integrity security without a birthday-bound restriction on the number of forgery attempts (decryption queries) by the adversary.

4.5 Specification

For the sake of simplicity, we make no distinction between the sets $\mathbb{F}_2^{a+b}$ and $\mathbb{F}_2^a \times \mathbb{F}_2^b$, we interpret those to be the same. The only difference is that we write elements of the second as tuples, while the members of the first set are bit strings corresponding to the concatenation of the two elements in the tuple. The empty bitstring is denoted ϵ. The algorithms assume the byte order to be little-endian.

The specification of the SPARKLE permutation and of its various instances is given in Section 4.5.1. Then, we use these permutations to specify the hash functions ESCH in Section 4.5.4 and the authenticated ciphers SCHWAEMM in Section 4.5.5.

We use "+" to denote the addition modulo 2^{32} and $\oplus$ to denote the XOR of two bitstrings of the same size.

4.5.1 The Sparkle Permutations

Our schemes for authenticated encryption and hashing employ the permutation family SPARKLE which we specify in the following. In particular, the SPARKLE family consists of the permutations SPARKLE256$_{n_s}$, SPARKLE384$_{n_s}$ and SPARKLE512$_{n_s}$ with block sizes of 256, 384, and 512 bit, respectively. The parameter n_s refers to the number of *steps* and a permutation can be defined for any $n_s \in \mathbb{N}$. The permutations are built using the following main components:

- The *ARX-box* Alzette [Bei+20b] (shortly denoted A), i.e., a 64-bit block cipher with a 32-bit key

$$A\colon (\mathbb{F}_2^{32} \times \mathbb{F}_2^{32}) \times \mathbb{F}_2^{32} \to (\mathbb{F}_2^{32} \times \mathbb{F}_2^{32}), ((x, y), c) \mapsto (u, v)\ .$$

  We define A_c to be the permutation $(x, y) \mapsto A(x, y, c)$ from $\mathbb{F}_2^{32} \times \mathbb{F}_2^{32}$ to $\mathbb{F}_2^{32} \times \mathbb{F}_2^{32}$.

- A linear *diffusion layer* $\mathcal{L}_{n_b}\colon \mathbb{F}_2^{64n_b} \to \mathbb{F}_2^{64n_b}$, where n_b denotes the number of 64-bit branches, i.e., the block size divided by 64. It is necessary that n_b is even.

The high-level structure of the permutations is given in Algorithms 1, 2 and 3, respectively. It is a classical Substitution-Permutation Network (SPN) construction except that functions playing the role of the S-boxes are different in each branch. More specifically, each member of the permutation family iterates a parallel application of Alzette under different, branch-dependent, constants c_i. This small 64-bit block cipher is specified in Section 4.5.2. It is followed by an application of $\mathcal{L}_{n_b}$, a linear permutation operating on all branches; it is specified in Section 4.5.3. We call such a parallel application of Alzette followed by the linear layer a *step*. The high-level structure of a step is represented in Figure 4.1. Before each step, a sparse step-dependent constant is XORed to the cipher's state (i.e., to y_0 and y_1).

A self-contained C implementation of the SPARKLE permutation, parameterized by the number of branches n_b and the number of steps n_s, can be found in Section 4.7. The implementation uses a single array named `state` of type `uint32_t` that consists of $2n_b$ elements to represent the state. More precisely, `state[0]` $= x_0$, `state[1]` $= y_0$, `state[2]` $= x_1$, `state[3]` $= y_1$, ... `state[2*nb-2]` $= x_{n_b-1}$, and `state[2*nb-1]` $=$

y_{n_b-1}. Each 32-bit word contains four state bytes in little-endian order. More precisely, if $(m_0, m_1, \dots, m_{n-1}) \in \mathbb{F}_2^n$, $n \in \{256, 384, 512\}$, is an input to a Sparkle instance, it is mapped to the state words via `state[`k`]` $=$

$$m_{32k+24} \| m_{32k+25} \| \cdots \| m_{32k+31} \| m_{32k+16} \| m_{32k+17} \| \cdots$$
$$m_{32k+23} \| \cdots \| m_{32k} \| m_{32k+1} \| \cdots \| m_{32k+7}$$

and the inverse mapping is used for transforming state words back to bitstrings.[3]

In what follows, we rely on the following definition given below to simplify our descriptions.

Definition 1 (Left/Right branches)**.** We call *left branches* those that correspond to the state inputs $(x_0, y_0), (x_1, y_1), \dots, (x_{n_b/2-1}, y_{n_b/2-1})$, and we call *right branches* those corresponding to $(x_{n_b/2}, y_{n_b/2}), \dots, (x_{n_b-2}, y_{n_b-2}), (x_{n_b-1}, y_{n_b-1})$.

Algorithm 1 Sparkle256$_{n_s}$
In/Out: $\big((x_0, y_0), ..., (x_3, y_3)\big), x_i, y_i \in \mathbb{F}_2^{32}$

```
$(c_0, c_1) \leftarrow$ (0xB7E15162,0xBF715880)
$(c_2, c_3) \leftarrow$ (0x38B4DA56,0x324E7738)
$(c_4, c_5) \leftarrow$ (0xBB1185EB,0x4F7C7B57)
$(c_6, c_7) \leftarrow$ (0xCFBFA1C8,0xC2B3293D)
for all $s \in [0, n_s - 1]$ do
    $y_0 \leftarrow y_0 \oplus c_{(s \bmod 8)}$
    $y_1 \leftarrow y_1 \oplus (s \bmod 2^{32})$
    for all $i \in [0, 3]$ do
        $(x_i, y_i) \leftarrow A_{c_i}(x_i, y_i)$
    end for
    $((x_0, y_0), ..., (x_3, y_3)) \leftarrow$
$\mathcal{L}_4((x_0, y_0), ..., (x_3, y_3))$
end for
return $((x_0, y_0), ..., (x_3, y_3))$
```

Algorithm 2 Sparkle384$_{n_s}$
In/Out: $\big((x_0, y_0), ..., (x_5, y_5)\big), x_i, y_i \in \mathbb{F}_2^{32}$

```
$(c_0, c_1) \leftarrow$ (0xB7E15162,0xBF715880)
$(c_2, c_3) \leftarrow$ (0x38B4DA56,0x324E7738)
$(c_4, c_5) \leftarrow$ (0xBB1185EB,0x4F7C7B57)
$(c_6, c_7) \leftarrow$ (0xCFBFA1C8,0xC2B3293D)
for all $s \in [0, n_s - 1]$ do
    $y_0 \leftarrow y_0 \oplus c_{(s \bmod 8)}$
    $y_1 \leftarrow y_1 \oplus (s \bmod 2^{32})$
    for all $i \in [0, 5]$ do
        $(x_i, y_i) \leftarrow A_{c_i}(x_i, y_i)$
    end for
    $((x_0, y_0), ..., (x_5, y_5)) \leftarrow$
$\mathcal{L}_6((x_0, y_0), ..., (x_5, y_5))$
end for
return $((x_0, y_0), ..., (x_5, y_5))$
```

Algorithm 3 Sparkle512$_{n_s}$
In/Out: $\big((x_0, y_0), ..., (x_7, y_7)\big), x_i \in \mathbb{F}_2^{32}, y_i \in \mathbb{F}_2^{32}$

```
$(c_0, c_1) \leftarrow$ (0xB7E15162,0xBF715880)
$(c_2, c_3) \leftarrow$ (0x38B4DA56,0x324E7738)
$(c_4, c_5) \leftarrow$ (0xBB1185EB,0x4F7C7B57)
$(c_6, c_7) \leftarrow$ (0xCFBFA1C8,0xC2B3293D)
for all $s \in [0, n_s - 1]$ do
    $y_0 \leftarrow y_0 \oplus c_{(s \bmod 8)}$
    $y_1 \leftarrow y_1 \oplus (s \bmod 2^{32})$
    for all $i \in [0, 7]$ do
        $(x_i, y_i) \leftarrow A_{c_i}(x_i, y_i)$
    end for
    $((x_0, y_0), ..., (x_7, y_7)) \leftarrow \mathcal{L}_8((x_0, y_0), ..., (x_7, y_7))$
end for
return $((x_0, y_0), ..., (x_7, y_7))$
```

[3]Note that the indirect injection through $\mathcal{M}_{h_b}$ in Esch also operates on state words. Therefore, the same mapping of bitstrings to words (and vice versa) is applied.

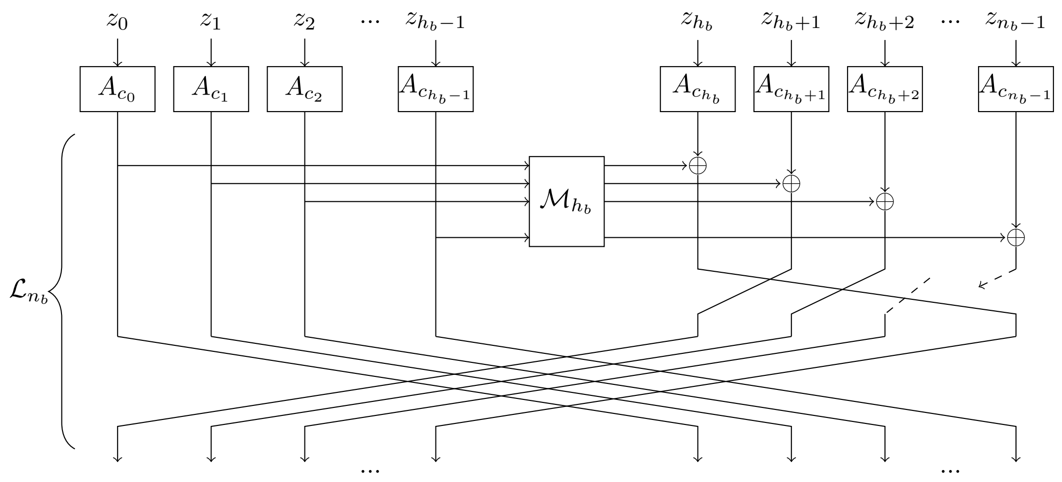

FIGURE 4.1: The overall structure of a step of SPARKLE. z_i denotes the 64-bit input (x_i, y_i) to the corresponding Alzette instance.

Specific Instances. The SPARKLE permutations are defined for 4, 6 and 8 branches and for any number of steps. Unlike in other sponge algorithms such as, e.g., SHA-3, we use two versions of the permutations which *differ only by the number of steps* used. More precisely, we use a *slim* and a *big* instance of SPARKLE. The slim and big versions of all SPARKLE instances are given in Table 4.1.

Name	n	# steps slim	# steps big
SPARKLE256	256	7	10
SPARKLE384	384	7	11
SPARKLE512	512	8	12

TABLE 4.1: The different versions of each SPARKLE instance.

4.5.2 The ARX-box Alzette

Alzette, shortly denoted A, is a 64-bit block cipher. It is specified in Algorithm 4 and depicted in Figure 4.2. It can be understood as a four-round iterated block cipher for which the rounds differ in the rotation amounts. After each round, the 32-bit constant (i.e., the key) is XORed to the left word. Note that, as Alzette has a simple Feistel-like structure, the computation of the inverse is straightforward.

Its purpose is to provide non-linearity to the whole permutation and to ensure a quick diffusion within each branch—the diffusion between the branches being ensured by the linear layer (Section 4.5.3). Its round constants ensure that the computations in each branch are independent from one another to break the symmetry of the permutation structure we chose. As the rounds themselves are different (because of different rotation amounts), we do not rely on the round constant to provide independence between the rounds of Alzette.

4.5.3 The Diffusion Layer

The diffusion layer has a structure which draws heavily from the one used in SPARX-128 [Din+]. We denote it $\mathcal{L}_{n_b}$. It is a Feistel round with a linear Feistel function $\mathcal{M}_{h_b}$ which permutes $\left(\mathbb{F}_2^{64}\right)^{h_b}$, where $h_b = \frac{n_b}{2}$. More formally, $\mathcal{M}_{h_b}$ is defined as follows.

Algorithm 4 A_c

Input/Output: $(x, y) \in \mathbb{F}_2^{32} \times \mathbb{F}_2^{32}$

```
  $x \leftarrow x + (y \ggg 31)$
  $y \leftarrow y \oplus (x \ggg 24)$
  $x \leftarrow x \oplus c$
  $x \leftarrow x + (y \ggg 17)$
  $y \leftarrow y \oplus (x \ggg 17)$
  $x \leftarrow x \oplus c$
  $x \leftarrow x + (y \ggg 0)$
  $y \leftarrow y \oplus (x \ggg 31)$
  $x \leftarrow x \oplus c$
  $x \leftarrow x + (y \ggg 24)$
  $y \leftarrow y \oplus (x \ggg 16)$
  $x \leftarrow x \oplus c$
  return $(x, y)$
```

Definition 2. Let $w > 1$ be an integer. We denote $\mathcal{M}_w$ the permutation of $(\mathbb{F}_2^{32})^w$ such that

$$\mathcal{M}_w\big((x_0, y_0), \ldots, (x_{w-1}, y_{w-1})\big) \;=\; \big((u_0, v_0), \ldots, (u_{w-1}, v_{w-1})\big)$$

where the branches (u_i, v_i) are obtained via the following equations

$$\begin{aligned} t_y &\leftarrow \bigoplus_{i=0}^{w-1} y_i \ , \ t_x \leftarrow \bigoplus_{i=0}^{w-1} x_i \ , \\ u_i &\leftarrow x_i \oplus \ell(t_y), \ \forall i \in \{0, ..., w-1\} \ , \\ v_i &\leftarrow y_i \oplus \ell(t_x), \ \forall i \in \{0, ..., w-1\} \ , \end{aligned} \tag{4.1}$$

where the indices are understood modulo w, and where $\ell : \mathbb{F}_2^{32} \rightarrow \mathbb{F}_2^{32}$ is a permutation defined by

$$\ell(x) = (x \lll 16) \oplus (x \& \texttt{0xffff}) \ ,$$

where $x\&y$ is a C-style notation denoting the bitwise AND of x and y. Note in particular that, if y and z are in $\mathbb{F}_2^{16}$ so that $y||z \in \mathbb{F}_2^{32}$, then

$$\ell(y||z) = z||(y \oplus z) \ .$$

The diffusion layer $\mathcal{L}_{n_b}$ then applies the corresponding Feistel function $\mathcal{M}_{h_b}$ and swaps the left branches with the right branches. However, before the branches are swapped, we rotate the branches on the right side by 1 branch to the left. This process is pictured in Figure 4.1. Algorithms describing the three diffusion layers used in our permutations are given in Algorithms 5, 6 and 7.

4.5.4 The Hash Functions Esch256 and Esch384

4.5.4.1 Instances

We propose two instances for hashing, i.e., Esch256 and Esch384, which allow to process messages $M \in \mathbb{F}_2^*$ of arbitrary length[4] and output a digest D of bitlengths 256,

[4]More rigorously, all bitlengths under a given (very large) threshold are supported.

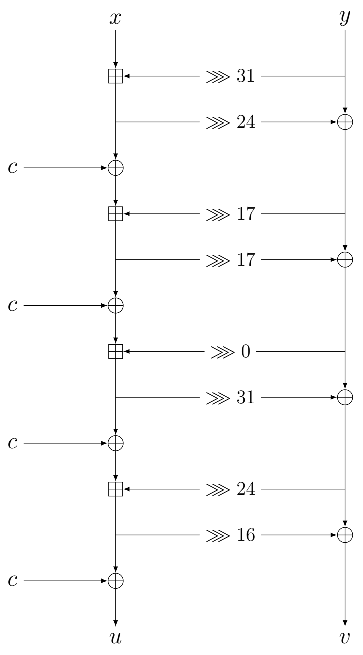

FIGURE 4.2: The structure of the Alzette instance A_c.

Algorithm 5 $\mathcal{L}_4$
Input/Output: $\big((x_0, y_0), (x_1, y_1), (x_2, y_2), (x_3, y_3)\big) \in (\mathbb{F}_2^{32} \times \mathbb{F}_2^{32})^4$

```
  // Feistel round
  $(t_x, t_y) \leftarrow \big(x_0 \oplus x_1, y_0 \oplus y_1\big)$
  $(t_x, t_y) \leftarrow \big((t_x \oplus (t_x \ll 16)) \lll 16,\ (t_y \oplus (t_y \ll 16)) \lll 16\big)$
  $(y_2, y_3) \leftarrow (y_2 \oplus y_0 \oplus t_x,\ y_3 \oplus y_1 \oplus t_x)$
  $(x_2, x_3) \leftarrow (x_2 \oplus x_0 \oplus t_y,\ x_3 \oplus x_1 \oplus t_y)$
  // Branch permutation
  $(x_0, x_1, x_2, x_3) \leftarrow (x_3, x_2, x_0, x_1)$
  $(y_0, y_1, y_2, y_3) \leftarrow (y_3, y_2, y_0, y_1)$
  return $\big((x_0, y_0), \ldots, (x_3, y_3)\big)$
```

and 384, respectively. Our primary member for hashing is ESCH256. They employ the well-known sponge construction, which is instantiated with SPARKLE permutations and parameterized by the rate r and the capacity c. The slim version is used during both absorption and squeezing. The big one is used in between the two phases. Table 4.2 gives an overview of the parameters used in the corresponding sponges. The maximum length is chosen as $r \times 2^{c/2}$ bits, where c is both the capacity and the digest size.

4.5.4.2 Specification of the Hash Functions

In both ESCH256 and ESCH384, the rate r is fixed to 128. This means that the message M has to be padded such that its length in bit becomes a multiple of 128. For this, we use the simple padding rule that appends 10^*. It is formalized in Algorithm 8 which describes how a block with length strictly smaller that r is turned into a block of length r.

The different digest sizes and the corresponding security levels are obtained using different permutation sizes in the sponge, i.e., SPARKLE384$_7$ and SPARKLE384$_{11}$ for

Algorithm 6 $\mathcal{L}_6$

Input/Output: $((x_0, y_0), \dots, (x_5, y_5)) \in (\mathbb{F}_2^{32} \times \mathbb{F}_2^{32})^6$

```
  // Feistel round
  $(t_x, t_y) \leftarrow (x_0 \oplus x_1 \oplus x_2,\ y_0 \oplus y_1 \oplus y_2)$
  $(t_x, t_y) \leftarrow ((t_x \oplus (t_x \ll 16)) \lll 16,\ (t_y \oplus (t_y \ll 16)) \lll 16)$
  $(y_3, y_4, y_5) \leftarrow (y_3 \oplus y_0 \oplus t_x,\ y_4 \oplus y_1 \oplus t_x,\ y_5 \oplus y_2 \oplus t_x)$
  $(x_3, x_4, x_5) \leftarrow (x_3 \oplus x_0 \oplus t_y,\ x_4 \oplus x_1 \oplus t_y,\ x_5 \oplus x_2 \oplus t_y)$
  // Branch permutation
  $(x_0, x_1, x_2, x_3, x_4, x_5) \leftarrow (x_4, x_5, x_3, x_0, x_1, x_2)$
  $(y_0, y_1, y_2, y_3, y_4, y_5) \leftarrow (y_4, y_5, y_3, y_0, y_1, y_2)$
  return $((x_0, y_0), \dots, (x_5, y_5))$
```

Algorithm 7 $\mathcal{L}_8$

Input/Output: $((x_0, y_0), \dots, (x_7, y_7)) \in (\mathbb{F}_2^{32} \times \mathbb{F}_2^{32})^8$

```
  // Feistel round
  $(t_x, t_y) \leftarrow (x_0 \oplus x_1 \oplus x_2 \oplus x_3, y_0 \oplus y_1 \oplus y_2 \oplus y_3)$
  $(t_x, t_y) \leftarrow ((t_x \oplus (t_x \ll 16)) \lll 16, (t_y \oplus (t_y \ll 16)) \lll 16)$
  $(y_4, y_5, y_6, y_7) \leftarrow (y_4 \oplus y_0 \oplus t_x,\ y_5 \oplus y_1 \oplus t_x,\ y_6 \oplus y_2 \oplus t_x,\ y_7 \oplus y_3 \oplus t_x)$
  $(x_4, x_5, x_6, x_7) \leftarrow (x_4 \oplus x_0 \oplus t_y,\ x_5 \oplus x_1 \oplus t_y,\ x_6 \oplus x_2 \oplus t_y,\ x_7 \oplus x_3 \oplus t_y)$
  // Branch permutation
  $(x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7) \leftarrow (x_5, x_6, x_7, x_4, x_0, x_1, x_2, x_3)$
  $(y_0, y_1, y_2, y_3, y_4, y_5, y_6, y_7) \leftarrow (y_5, y_6, y_7, y_4, y_0, y_1, y_2, y_3)$
  return $((x_0, y_0), \dots, (x_7, y_7))$
```

	n	r	c	collision	2nd preimage	preimage	data limit (bytes)
Esch256	384	128	256	128	128	128	2^{132}
Esch384	512	128	384	192	192	192	2^{196}
XOEsch256	384	128	256	$\min\{128, \frac{t}{2}\}$	$\min\{128, t\}$	$\min\{128, t\}$	2^{132}
XOEsch384	512	128	384	$\min\{192, \frac{t}{2}\}$	$\min\{192, t\}$	$\min\{192, t\}$	2^{196}

Table 4.2: The hashing instances with their security level in bit with regard to collision resistance and (second) preimage resistance and the limitation on the message size in bytes. For the security levels of the XOFs, we assume that t is smaller than the allowed data limit. The first line refers to our primary member, i.e. Esch256.

Esch256 and Sparkle512$_8$ and Sparkle512$_{12}$ for Esch384. The algorithms are formally specified in Algorithm 9 and 10 and are depicted in Figure 4.3 and Figure 4.4, respectively. Note that the 128 bits of message blocks are injected *indirectly*, i.e., they are first padded with zeros and transformed via $\mathcal{M}_3$ in Esch256, resp., $\mathcal{M}_4$ in Esch384, and the resulting image is XORed to the *leftmost* branches of the state. We stress that this tweak can still be expressed in the regular sponge mode. Instead of injecting the messages through $\mathcal{M}_{h_b}$, one can use an equivalent representation in which the message is injected as usual and the permutation is defined by prepending $\mathcal{M}_{h_b}$ and appending $\mathcal{M}_{h_b}^{-1}$ to Sparkle$_{n_b}$.

For generating the digest, we use the simple truncation function trunc_t which returns the t leftmost bits of the internal state.

A message with a length that is a multiple of r is not padded. To prevent trivial collisions, we borrow the technique introduced in [Hir16] and xor $\mathsf{Const}_\mathsf{M}$ to the inner

Algorithm 8 pad_r

Input/Output: $M \in \mathbb{F}_2^*$, with $|M| < r$

```
$i \leftarrow (-|M| - 1) \mod r$
$M \leftarrow M\|1\|0^i$
**return** $M$
```

part, where $\mathsf{Const_M}$ is different depending on whether the message was padded or not.

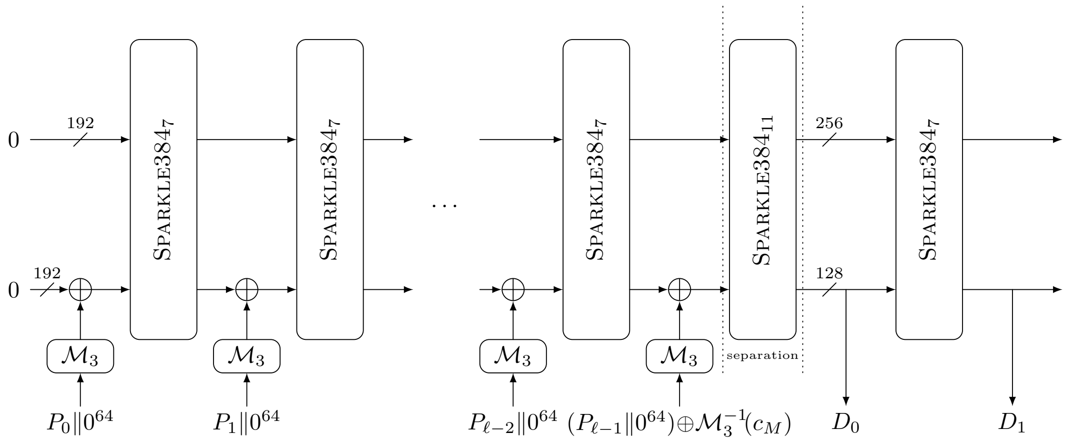

FIGURE 4.3: The Hash Function ESCH256 with rate $r = 128$ and capacity $c = 256$. The constant c_M is equal to $(0, 0, \ldots, 0, 1) \in \mathbb{F}_2^{192}$ if the last block was padded and equal to $(0, 0, \ldots, 0, 1, 0) \in \mathbb{F}_2^{192}$ otherwise.

Algorithm 9 ESCH256

Input: $M \in \mathbb{F}_2^*$ *Output:* $D \in \mathbb{F}_2^{256}$

```
    // Padding the message
    if $M \neq \epsilon$ then
        $P_0 \| P_1 \| \dots \| P_{\ell-1} \leftarrow M$
    with $\forall i < \ell - 1 \colon |P_i| = 128$ and $1 \leq |P_{\ell-1}| \leq 128$
    else
        $\ell \leftarrow 1$
        $P_0 \leftarrow \epsilon$
    end if
    if $|P_{\ell-1}| < 128$ then
        $P_{\ell-1} \leftarrow \mathsf{pad}_{128}(P_{\ell-1})$
        $\mathsf{Const_M} \leftarrow (1 \lll 192)$
    else
        $\mathsf{Const_M} \leftarrow (2 \lll 192)$
    end if
    // Absorption
    $S \leftarrow 0 \in \mathbb{F}_2^{384}$
    for all $j = 0, \dots, \ell - 2$ do
        $P'_j \leftarrow \mathcal{M}_3(P_j \| 0^{64})$
        $S \leftarrow \text{SPARKLE384}_7\big(S \oplus (P'_j \| 0^{192})\big)$
    end for
    $P'_{\ell-1} \leftarrow \mathcal{M}_3(P_{\ell-1} \| 0^{64})$
    $S \leftarrow \text{SPARKLE384}_{11}\big(S \oplus (P'_{\ell-1} \| 0^{192}) \oplus \mathsf{Const_M}\big)$
    // Squeezing
    $D_0 \leftarrow \mathsf{trunc}_{128}(S)$
    $S \leftarrow \text{SPARKLE384}_7(S)$
    $D_1 \leftarrow \mathsf{trunc}_{128}(S)$

    return $D_0 \| D_1$
```

Algorithm 10 ESCH384

Input: $M \in \mathbb{F}_2^*$ *Output:* $D \in \mathbb{F}_2^{384}$

```
    // Padding the message
    if $M \neq \epsilon$ then
        $P_0 \| P_1 \| \dots \| P_{\ell-1} \leftarrow M$
    with $\forall i < \ell - 1 \colon |P_i| = 128$ and $1 \leq |P_{\ell-1}| \leq 128$
    else
        $\ell \leftarrow 1$
        $P_0 \leftarrow \epsilon$
    end if
    if $|P_{\ell-1}| < 128$ then
        $P_{\ell-1} \leftarrow \mathsf{pad}_{128}(P_{\ell-1})$
        $\mathsf{Const_M} \leftarrow (1 \lll 256)$
    else
        $\mathsf{Const_M} \leftarrow (2 \lll 256)$
    end if
    // Absorption
    $S \leftarrow 0 \in \mathbb{F}_2^{512}$
    for all $j = 0, \dots, \ell - 2$ do
        $P'_j \leftarrow \mathcal{M}_4(P_j \| 0^{128})$
        $S \leftarrow \text{SPARKLE512}_8\big(S \oplus (P'_j \| 0^{256})\big)$
    end for
    $P'_{\ell-1} \leftarrow \mathcal{M}_4(P_{\ell-1} \| 0^{128})$
    $S \leftarrow \text{SPARKLE512}_{12}\big(S \oplus (P'_{\ell-1} \| 0^{256}) \oplus \mathsf{Const_M}\big)$
    // Squeezing
    $D_0 \leftarrow \mathsf{trunc}_{128}(S)$
    $S \leftarrow \text{SPARKLE512}_8(S)$
    $D_1 \leftarrow \mathsf{trunc}_{128}(S)$
    $S \leftarrow \text{SPARKLE512}_8(S)$
    $D_2 \leftarrow \mathsf{trunc}_{128}(S)$
    return $D_0 \| D_1 \| D_2$
```

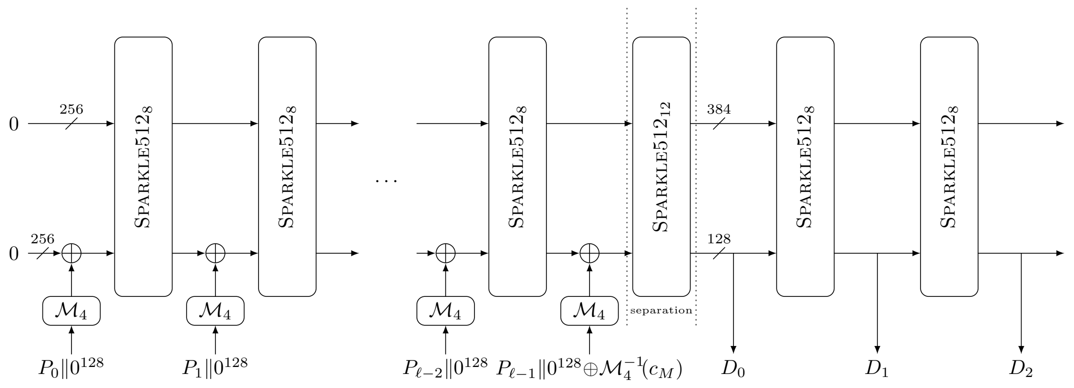

FIGURE 4.4: The Hash Function ESCH384 with rate $r = 128$ and capacity $c = 384$. The constant c_M is equal to $(0, 0, \dots, 0, 1) \in \mathbb{F}_2^{256}$ if the last block was padded and equal to $(0, 0, \dots, 0, 1, 0) \in \mathbb{F}_2^{256}$ otherwise.

4.5.4.3 The Extendable-Output Functions XOEsch256 and XOEsch384

The hash functions Esch256 and Esch384 can easily be adapted to provide outputs of arbitrary length. We define the extendable-output functions (XOFs) XOEsch256 and XOEsch384, which are very similar to their hashing counterparts. Besides that other values for the constants $\mathsf{Const}_\mathsf{M}$ are used in order to separate between the different use-cases, the only difference is that the XOFs obtain an additional input parameter t which defines the size of the output string. The squeezing phase is extended in order to provide the output of the required length. XOEsch256 and XOEsch384 are formally described in Algorithms 11 and 12, respectively. The parameters and security levels are given in Table 4.2.

Algorithm 11 XOEsch256

Input: $M \in \mathbb{F}_2^*, t \in \mathbb{N}$ *Output:* $D \in \mathbb{F}_2^t$

```
// Padding the message
if $M \neq \epsilon$ then
    $P_0 \| P_1 \| \dots \| P_{\ell-1} \leftarrow M$
with $\forall i < \ell - 1 \colon |P_i| = 128$ and $1 \le |P_{\ell-1}| \le 128$
else
    $\ell \leftarrow 1$
    $P_0 \leftarrow \epsilon$
end if
if $|P_{\ell-1}| < 128$ then
    $P_{\ell-1} \leftarrow \mathsf{pad}_{128}(P_{\ell-1})$
    $\mathsf{Const}_\mathsf{M} \leftarrow (1 \ll 192) \oplus (4 \ll 192)$
else
    $\mathsf{Const}_\mathsf{M} \leftarrow (2 \ll 192) \oplus (4 \ll 192)$
end if
// Absorption
$S \leftarrow 0 \in \mathbb{F}_2^{384}$
for all $j = 0, \dots, \ell - 2$ do
    $P'_j \leftarrow \mathcal{M}_3(P_j \| 0^{64})$
    $S \leftarrow \text{SPARKLE384}_7\big(S \oplus (P'_j \| 0^{192})\big)$
end for
$P'_{\ell-1} \leftarrow \mathcal{M}_3(P_{\ell-1} \| 0^{64})$
$S \leftarrow \text{SPARKLE384}_{11}\big(S \oplus (P'_{\ell-1} \| 0^{192}) \oplus \mathsf{Const}_\mathsf{M}\big)$
// Squeezing
$D_0 \leftarrow \mathsf{trunc}_{128}(S)$
for all $j = 1, \dots, \lceil t/128 \rceil - 1$ do
    $S \leftarrow \text{SPARKLE384}_7\big(S\big)$
    $D_j \leftarrow \mathsf{trunc}_{128}(S)$
end for
return $\mathsf{trunc}_t(D_0 \| D_1 \| \dots \| D_{\lceil t/128 \rceil - 1})$
```

Algorithm 12 XOEsch384

Input: $M \in \mathbb{F}_2^*, t \in \mathbb{N}$ *Output:* $D \in \mathbb{F}_2^t$

```
// Padding the message
if $M \neq \epsilon$ then
    $P_0 \| P_1 \| \dots \| P_{\ell-1} \leftarrow M$
with $\forall i < \ell - 1 \colon |P_i| = 128$ and $1 \le |P_{\ell-1}| \le 128$
else
    $\ell \leftarrow 1$
    $P_0 \leftarrow \epsilon$
end if
if $|P_{\ell-1}| < 128$ then
    $P_{\ell-1} \leftarrow \mathsf{pad}_{128}(P_{\ell-1})$
    $\mathsf{Const}_\mathsf{M} \leftarrow (1 \ll 256) \oplus (4 \ll 256)$
else
    $\mathsf{Const}_\mathsf{M} \leftarrow (2 \ll 256) \oplus (4 \ll 256)$
end if
// Absorption
$S \leftarrow 0 \in \mathbb{F}_2^{512}$
for all $j = 0, \dots, \ell - 2$ do
    $P'_j \leftarrow \mathcal{M}_4(P_j \| 0^{128})$
    $S \leftarrow \text{SPARKLE512}_8\big(S \oplus (P'_j \| 0^{256})\big)$
end for
$P'_{\ell-1} \leftarrow \mathcal{M}_4(P_{\ell-1} \| 0^{128})$
$S \leftarrow \text{SPARKLE512}_{12}\big(S \oplus (P'_{\ell-1} \| 0^{256}) \oplus \mathsf{Const}_\mathsf{M}\big)$
// Squeezing
$D_0 \leftarrow \mathsf{trunc}_{128}(S)$
for all $j = 1, \dots, \lceil t/128 \rceil - 1$ do
    $S \leftarrow \text{SPARKLE512}_8\big(S\big)$
    $D_j \leftarrow \mathsf{trunc}_{128}(S)$
end for
return $\mathsf{trunc}_t(D_0 \| D_1 \| \dots \| D_{\lceil t/128 \rceil - 1})$
```

4.5.5 The Authenticated Cipher Family Schwaemm

4.5.5.1 Instances

We propose four instances for authenticated encryption with associated data, i.e. Schwaemm128-128, Schwaemm256-128, Schwaemm192-192 and Schwaemm256-256 which, for a given key K and nonce N allow to process associated data A and messages M of arbitrary length[5] and output a ciphertext C with $|C| = |M|$ and an authentication tag T. For given (K, N, A, C, T), the decryption procedure returns the decryption M of C if the tag T is valid, otherwise it returns the error symbol $\perp$. Our primary member of the family is Schwaemm256-128. All instances use (a slight variation of) the Beetle mode of operation presented in [Cha+18], which is based on the well-known SpongeWrap AEAD mode [Ber+11a]. The difference between the instances is the version of the underlying Sparkle permutation (and thus the rate and capacity is different) and the size of the authentication tag. As a naming convention, we used Schwaemmr-c, where r refers to the size of the rate and c to the size of the capacity in bits. Similar as for hashing, we use the big version of Sparkle for initialization, separation between processing of associated data and secret message, and finalization, and the slim version of Sparkle for updating the intermediate state otherwise. Table 4.3 gives an overview of the parameters of the Schwaemm instances. The data limits correspond to 2^{64} blocks of r bits rounded up to the closest power of two, except for the high security Schwaemm256-256 for which it is $r \times 2^{128}$ bits.

	n	r	c	$\lvert K\rvert$	$\lvert N\rvert$	$\lvert T\rvert$	security	data limit (bytes)
Schwaemm256-128	384	256	128	128	256	128	120	2^{68}
Schwaemm192-192	384	192	192	192	192	192	184	2^{68}
Schwaemm128-128	256	128	128	128	128	128	120	2^{68}
Schwaemm256-256	512	256	256	256	256	256	248	2^{133}

Table 4.3: The instances we provide for authenticated encryption together with their (joint) security level in bit with regard to confidentiality and integrity and the limitation in the data (in bytes) to be processed. The first line refers to our primary member, i.e. Schwaemm256-128.

4.5.5.2 The Algorithms

The main difference between the Beetle mode and duplexed sponge modes is the usage of a combined feedback ρ to differentiate the ciphertext blocks and the outer part of the states. This combined feedback is created by applying the function FeistelSwap to the outer part of the state, which is computed as

$$\mathsf{FeistelSwap}(S) = S_2 \| (S_2 \oplus S_1) \; ,$$

where $S \in \mathbb{F}_2^r$ and $S_1 \| S_2 = S$ with $|S_1| = |S_2| = \frac{r}{2}$. The feedback function $\rho \colon (\mathbb{F}_2^r \times \mathbb{F}_2^r) \to (\mathbb{F}_2^r \times \mathbb{F}_2^r)$ is defined as $\rho(S, D) = (\rho_1(S, D), \rho_2(S, D))$, where

$$\rho_1 \colon (S, D) \mapsto \mathsf{FeistelSwap}(S) \oplus D, \quad \rho_2 \colon (S, D) \mapsto S \oplus D \; .$$

[5] As for the hash function, the length can be chosen arbitrarily but it has do be under thresholds that are given in Table 4.3.

For decryption, we have to use the inverse feedback function $\rho' \colon (\mathbb{F}_2^r \times \mathbb{F}_2^r) \to (\mathbb{F}_2^r \times \mathbb{F}_2^r)$ defined as $\rho'(S, D) = (\rho'_1(S, D), \rho'_2(S, D))$, where

$$\rho'_1 \colon (S, D) \mapsto \mathsf{FeistelSwap}(S) \oplus S \oplus D, \quad \rho'_2 \colon (S, D) \mapsto S \oplus D\ .$$

After each application of ρ and the additions of the domain separation constants, i.e., before each call to the Sparkle permutation except the one for initialization, we prepend a *rate whitening* layer which XORs the value of $\mathcal{W}_{c,r}(S_R)$ to the outer part, where S_R denotes the internal state corresponding to the inner part. For the Schwaemm instances with $r = c$, we define $\mathcal{W}_{c,r} \colon \mathbb{F}_2^c \to \mathbb{F}_2^r$ as the identity (i.e., we just XOR the inner part to the outer part). For Schwaemm256-128, we define $\mathcal{W}_{128,256}(x, y) = (x, y, x, y)$, where $x, y \in \mathbb{F}_2^{64}$. Note that this tweak can still be described in the Beetle framework as the prepended rate whitening can be considered to be part of the definition of the underlying permutation.

Figure 4.5 depicts the mode for our primary member Schwaemm256-128. The formal specifications of the encryption and decryption procedures of the four family members are given in Algorithms 13-20.

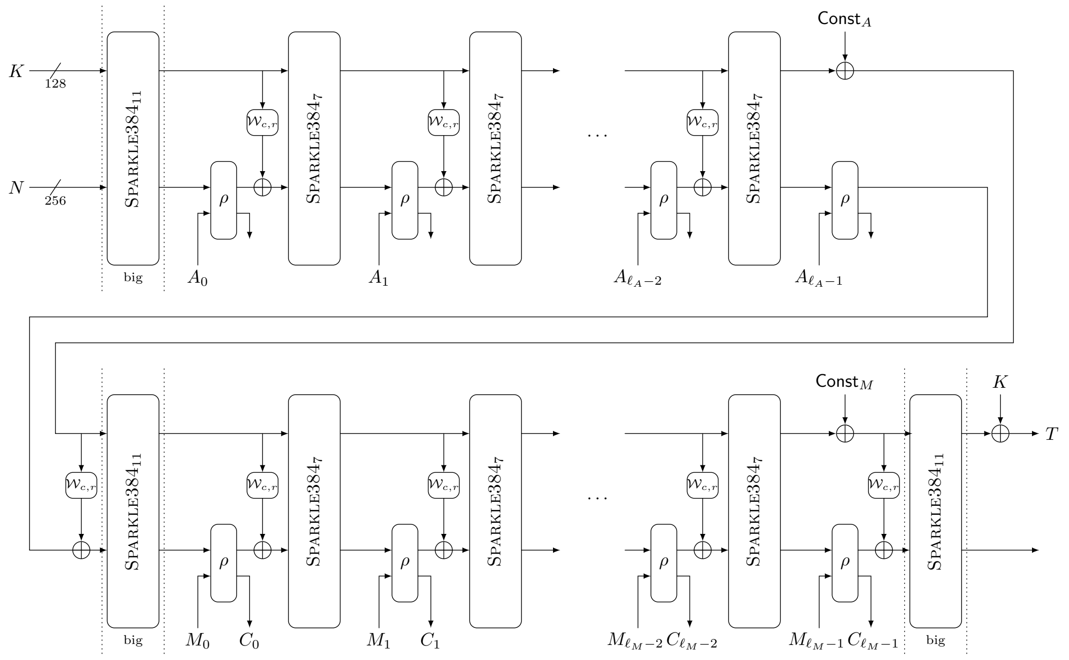

Figure 4.5: The Authenticated Encryption Algorithm Schwaemm256-128 with rate $r = 256$ and capacity $c = 128$.

Algorithm 13 Schwaemm256-128-Enc

Input: (K, N, A, M) where $K \in \mathbb{F}_2^{128}$ is a key, $N \in \mathbb{F}_2^{256}$ is a nonce and $A, M \in \mathbb{F}_2^*$

Output: (C, T), where $C \in \mathbb{F}_2^*$ is the ciphertext and $T \in \mathbb{F}_2^{128}$ is the authentication tag

```
 // Padding the associated data and message
if $A \neq \epsilon$ then
    $A_0 \| A_1 \| \dots \| A_{\ell_A - 1} \leftarrow A$ with $\forall i \in \{0, \dots, \ell_A - 2\} : |A_i| = 256$ and $1 \leq |A_{\ell_A - 1}| \leq 256$
    if $|A_{\ell_A - 1}| < 256$ then
        $A_{\ell_A - 1} \leftarrow \mathsf{pad}_{256}(A_{\ell_A - 1})$
        $\mathsf{Const}_A \leftarrow 0 \oplus (1 \ll 2)$
    else
        $\mathsf{Const}_A \leftarrow 1 \oplus (1 \ll 2)$
    end if
end if
if $M \neq \epsilon$ then
    $M_0 \| M_1 \| \dots \| M_{\ell_M - 1} \leftarrow M$ with $\forall i \in \{0, \dots, \ell_M - 2\} : |M_i| = 256$ and $1 \leq |M_{\ell_M - 1}| \leq 256$
    $t \leftarrow |M_{\ell_M - 1}|$
    if $|M_{\ell_M - 1}| < 256$ then
        $M_{\ell_M - 1} \leftarrow \mathsf{pad}_{256}(M_{\ell_M - 1})$
        $\mathsf{Const}_M \leftarrow 2 \oplus (1 \ll 2)$
    else
        $\mathsf{Const}_M \leftarrow 3 \oplus (1 \ll 2)$
    end if
end if
 // State initialization
$S_L \| S_R \leftarrow \text{SPARKLE384}_{11}\big(N \| K\big)$ with $|S_L| = 256$ and $|S_R| = 128$
 // Processing of associated data
if $A \neq \epsilon$ then
    for all $j = 0, \dots, \ell_A - 2$ do
        $S_L \| S_R \leftarrow \text{SPARKLE384}_{7}\big((\rho_1(S_L, A_j) \oplus \mathcal{W}_{128,256}(S_R)) \| S_R\big)$
    end for
     // Finalization if message is empty
    $S_L \| S_R \leftarrow \text{SPARKLE384}_{11}\big((\rho_1(S_L, A_{\ell_A - 1}) \oplus \mathcal{W}_{128,256}(S_R \oplus \mathsf{Const}_A)) \| (S_R \oplus \mathsf{Const}_A)\big)$
end if
 // Encrypting
if $M \neq \epsilon$ then
    for all $j = 0, \dots, \ell_M - 2$ do
        $C_j \leftarrow \rho_2(S_L, M_j)$
        $S_L \| S_R \leftarrow \text{SPARKLE384}_{7}\big((\rho_1(S_L, M_j) \oplus \mathcal{W}_{128,256}(S_R)) \| S_R\big)$
    end for
    $C_{\ell_M - 1} \leftarrow \mathsf{trunc}_t\big(\rho_2(S_L, M_{\ell_M - 1})\big)$
     // Finalization
    $S_L \| S_R \leftarrow \text{SPARKLE384}_{11}\big((\rho_1(S_L, M_{\ell_M - 1}) \oplus \mathcal{W}_{128,256}(S_R \oplus \mathsf{Const}_M)) \| (S_R \oplus \mathsf{Const}_M)\big)$
end if

return $(C_0 \| C_1 \| \dots \| C_{\ell_M - 1}, S_R \oplus K)$
```

Algorithm 14 SCHWAEMM256-128-DEC
Input: (K, N, A, C, T) where $K \in \mathbb{F}_2^{128}$ is a key, $N \in \mathbb{F}_2^{256}$ is a nonce, $A, C \in \mathbb{F}_2^*$ and $T \in \mathbb{F}_2^{128}$
Output: Decryption M of C if the tag T is valid, $\perp$ otherwise

```
if $A \neq \epsilon$ then
    $A_0 \| A_1 \| \dots \| A_{\ell_A - 1} \leftarrow A$ with $\forall i \in \{0, \dots, \ell_A - 2\} : |A_i| = 256$ and $1 \leq |A_{\ell_A - 1}| \leq 256$
    if $|A_{\ell_A - 1}| < 256$ then
        $A_{\ell_A - 1} \leftarrow \mathsf{pad}_{256}(A_{\ell_A - 1})$
        $\mathsf{Const}_A \leftarrow 0 \oplus (1 \ll 2)$
    else
        $\mathsf{Const}_A \leftarrow 1 \oplus (1 \ll 2)$
    end if
end if
if $C \neq \epsilon$ then
    $C_0 \| C_1 \| \dots \| C_{\ell_M - 1} \leftarrow C$ with $\forall i \in \{0, \dots, \ell_M - 2\} : |C_i| = 256$ and $1 \leq |C_{\ell_M - 1}| \leq 256$
    $t \leftarrow |C_{\ell_M - 1}|$
    if $|C_{\ell_M - 1}| < 256$ then
        $C_{\ell_M - 1} \leftarrow \mathsf{pad}_{256}(C_{\ell_M - 1})$
        $\mathsf{Const}_M \leftarrow 2 \oplus (1 \ll 2)$
    else
        $\mathsf{Const}_M \leftarrow 3 \oplus (1 \ll 2)$
    end if
end if
// State initialization
$S_L \| S_R \leftarrow \textsc{Sparkle384}_{11}(N \| K)$ with $|S_L| = 256$ and $|S_R| = 128$
// Processing of associated data
if $A \neq \epsilon$ then
    for all $j = 0, \dots, \ell_A - 2$ do
        $S_L \| S_R \leftarrow \textsc{Sparkle384}_7((\rho_1(S_L, A_j) \oplus \mathcal{W}_{128,256}(S_R)) \| S_R)$
    end for
    // Finalization if ciphertext is empty
    $S_L \| S_R \leftarrow \textsc{Sparkle384}_{11}((\rho_1(S_L, A_{\ell_A - 1}) \oplus \mathcal{W}_{128,256}(S_R \oplus \mathsf{Const}_A)) \| (S_R \oplus \mathsf{Const}_A))$
end if
// Decrypting
if $C \neq \epsilon$ then
    for all $j = 0, \dots, \ell_M - 2$ do
        $M_j \leftarrow \rho'_2(S_L, C_j)$
        $S_L \| S_R \leftarrow \textsc{Sparkle384}_7((\rho'_1(S_L, C_j) \oplus \mathcal{W}_{128,256}(S_R)) \| S_R)$
    end for
    $M_{\ell_M - 1} \leftarrow \mathsf{trunc}_t(\rho'_2(S_L, C_{\ell_M - 1}))$
    // Finalization and tag verification
    if $t < 256$ then
        $S_L \| S_R \leftarrow \textsc{Sparkle384}_{11}((\rho_1(S_L, \mathsf{pad}_{256}(M_{\ell_M - 1})) \oplus \mathcal{W}_{128,256}(S_R \oplus \mathsf{Const}_M)) \| (S_R \oplus \mathsf{Const}_M))$
    else
        $S_L \| S_R \leftarrow \textsc{Sparkle384}_{11}((\rho'_1(S_L, C_{\ell_M - 1}) \oplus \mathcal{W}_{128,256}(S_R \oplus \mathsf{Const}_M)) \| (S_R \oplus \mathsf{Const}_M))$
    end if
end if
if $S_R \oplus K = T$ then
    return $(M_0 \| M_1 \| \dots \| M_{\ell_M - 1})$
else
    return $\perp$
end if
```

Algorithm 15 Schwaemm192-192-Enc
Input: (K, N, A, M) where $K \in \mathbb{F}_2^{192}$ is a key, $N \in \mathbb{F}_2^{192}$ is a nonce and $A, M \in \mathbb{F}_2^*$
Output: (C, T), where $C \in \mathbb{F}_2^*$ is the ciphertext and $T \in \mathbb{F}_2^{192}$ is the authentication tag

```
 // Padding the associated data and message
if $A \neq \epsilon$ then
    $A_0 \| A_1 \| \ldots \| A_{\ell_A-1} \leftarrow A$ with $\forall i \in \{0, \ldots, \ell_A - 2\} : |A_i| = 192$ and $1 \leq |A_{\ell_A-1}| \leq 192$
    if $|A_{\ell_A-1}| < 192$ then
        $A_{\ell_A-1} \leftarrow \mathsf{pad}_{192}(A_{\ell_A-1})$
        $\mathsf{Const}_A \leftarrow 0 \oplus (1 \ll 3)$
    else
        $\mathsf{Const}_A \leftarrow 1 \oplus (1 \ll 3)$
    end if
end if
if $M \neq \epsilon$ then
    $M_0 \| M_1 \| \ldots \| M_{\ell_M-1} \leftarrow M$ with $\forall i \in \{0, \ldots, \ell_M - 2\} : |M_i| = 192$ and $1 \leq |M_{\ell_M-1}| \leq 192$
    $t \leftarrow |M_{\ell_M-1}|$
    if $|M_{\ell_M-1}| < 192$ then
        $M_{\ell_M-1} \leftarrow \mathsf{pad}_{192}(M_{\ell_M-1})$
        $\mathsf{Const}_M \leftarrow 2 \oplus (1 \ll 3)$
    else
        $\mathsf{Const}_M \leftarrow 3 \oplus (1 \ll 3)$
    end if
end if
 // State initialization
$S_L \| S_R \leftarrow \text{Sparkle384}_{11}(N \| K)$ with $|S_L| = 192$ and $|S_R| = 192$
 // Processing of associated data
if $A \neq \epsilon$ then
    for all $j = 0, \ldots, \ell_A - 2$ do
        $S_L \| S_R \leftarrow \text{Sparkle384}_7((\rho_1(S_L, A_j) \oplus S_R) \| S_R)$
    end for
     // Finalization if message is empty
    $S_L \| S_R \leftarrow \text{Sparkle384}_{11}((\rho_1(S_L, A_{\ell_A-1}) \oplus S_R \oplus \mathsf{Const}_A) \| (S_R \oplus \mathsf{Const}_A))$
end if
 // Encrypting
if $M \neq \epsilon$ then
    for all $j = 0, \ldots, \ell_M - 2$ do
        $C_j \leftarrow \rho_2(S_L, M_j)$
        $S_L \| S_R \leftarrow \text{Sparkle384}_7((\rho_1(S_L, M_j) \oplus S_R) \| S_R)$
    end for
    $C_{\ell_M-1} \leftarrow \mathsf{trunc}_t(\rho_2(S_L, M_{\ell_M-1}))$
     // Finalization
    $S_L \| S_R \leftarrow \text{Sparkle384}_{11}((\rho_1(S_L, M_{\ell_M-1}) \oplus S_R \oplus \mathsf{Const}_M) \| (S_R \oplus \mathsf{Const}_M))$
end if

return $(C_0 \| C_1 \| \ldots \| C_{\ell_M-1}, S_R \oplus K)$
```

Algorithm 16 Schwaemm192-192-Dec
Input: (K, N, A, C, T) where $K \in \mathbb{F}_2^{192}$ is a key, $N \in \mathbb{F}_2^{192}$ is a nonce, $A, C \in \mathbb{F}_2^*$ and $T \in \mathbb{F}_2^{192}$
Output: Decryption M of C if the tag T is valid, $\perp$ otherwise

```
if $A \neq \epsilon$ then
    $A_0 \| A_1 \| \dots \| A_{\ell_A - 1} \leftarrow A$ with $\forall i \in \{0, \dots, \ell_A - 2\} : |A_i| = 192$ and $1 \leq |A_{\ell_A - 1}| \leq 192$
    if $|A_{\ell_A - 1}| < 192$ then
        $A_{\ell_A - 1} \leftarrow \mathsf{pad}_{192}(A_{\ell_A - 1})$
        $\mathsf{Const}_A \leftarrow 0 \oplus (1 \ll 3)$
    else
        $\mathsf{Const}_A \leftarrow 1 \oplus (1 \ll 3)$
    end if
end if
if $C \neq \epsilon$ then
    $C_0 \| C_1 \| \dots \| C_{\ell_M - 1} \leftarrow C$ with $\forall i \in \{0, \dots, \ell_M - 2\} : |C_i| = 192$ and $1 \leq |C_{\ell_M - 1}| \leq 192$
    $t \leftarrow |C_{\ell_M - 1}|$
    if $|C_{\ell_M - 1}| < 192$ then
        $C_{\ell_M - 1} \leftarrow \mathsf{pad}_{192}(C_{\ell_M - 1})$
        $\mathsf{Const}_M \leftarrow 2 \oplus (1 \ll 3)$
    else
        $\mathsf{Const}_M \leftarrow 3 \oplus (1 \ll 3)$
    end if
end if
// State initialization
$S_L \| S_R \leftarrow \textsc{Sparkle384}_{11}\big(N \| K\big)$ with $|S_L| = 192$ and $|S_R| = 192$
// Processing of associated data
if $A \neq \epsilon$ then
    for all $j = 0, \dots, \ell_A - 2$ do
        $S_L \| S_R \leftarrow \textsc{Sparkle384}_7\big((\rho_1(S_L, A_j) \oplus S_R) \| S_R\big)$
    end for
    // Finalization if ciphertext is empty
    $S_L \| S_R \leftarrow \textsc{Sparkle384}_{11}\big((\rho_1(S_L, A_{\ell_A - 1}) \oplus S_R \oplus \mathsf{Const}_A) \| (S_R \oplus \mathsf{Const}_A)\big)$
end if
// Decrypting
if $C \neq \epsilon$ then
    for all $j = 0, \dots, \ell_M - 2$ do
        $M_j \leftarrow \rho'_2(S_L, C_j)$
        $S_L \| S_R \leftarrow \textsc{Sparkle384}_7\big((\rho'_1(S_L, C_j) \oplus S_R) \| S_R\big)$
    end for
    $M_{\ell_M - 1} \leftarrow \mathsf{trunc}_t\big(\rho'_2(S_L, C_{\ell_M - 1})\big)$
    // Finalization and tag verification
    if $t < 192$ then
        $S_L \| S_R \leftarrow \textsc{Sparkle384}_{11}\big((\rho_1(S_L, \mathsf{pad}_{192}(M_{\ell_M - 1})) \oplus S_R \oplus \mathsf{Const}_M) \| (S_R \oplus \mathsf{Const}_M)\big)$
    else
        $S_L \| S_R \leftarrow \textsc{Sparkle384}_{11}\big((\rho'_1(S_L, C_{\ell_M - 1}) \oplus S_R \oplus \mathsf{Const}_M) \| (S_R \oplus \mathsf{Const}_M)\big)$
    end if
end if
if $S_R \oplus K = T$ then
    return $(M_0 \| M_1 \| \dots \| M_{\ell_M - 1})$
else
    return $\perp$
end if
```

Algorithm 17 Schwaemm128-128-Enc
Input: (K, N, A, M) where $K \in \mathbb{F}_2^{128}$ is a key, $N \in \mathbb{F}_2^{128}$ is a nonce and $A, M \in \mathbb{F}_2^*$
Output: (C, T), where $C \in \mathbb{F}_2^*$ is the ciphertext and $T \in \mathbb{F}_2^{128}$ is the authentication tag

```
 // Padding the associated data and message
if $A \neq \epsilon$ then
    $A_0 \| A_1 \| \dots \| A_{\ell_A - 1} \leftarrow A$ with $\forall i \in \{0, \dots, \ell_A - 2\} : |A_i| = 128$ and $1 \leq |A_{\ell_A - 1}| \leq 128$
    if $|A_{\ell_A-1}| < 128$ then
        $A_{\ell_A-1} \leftarrow \mathsf{pad}_{128}(A_{\ell_A-1})$
        $\mathsf{Const}_A \leftarrow 0 \oplus (1 \ll 2)$
    else
        $\mathsf{Const}_A \leftarrow 1 \oplus (1 \ll 2)$
    end if
end if
if $M \neq \epsilon$ then
    $M_0 \| M_1 \| \dots \| M_{\ell_M - 1} \leftarrow M$ with $\forall i \in \{0, \dots, \ell_M - 2\} : |M_i| = 128$ and $1 \leq |M_{\ell_M - 1}| \leq 128$
    $t \leftarrow |M_{\ell_M-1}|$
    if $|M_{\ell_M-1}| < 128$ then
        $M_{\ell_M-1} \leftarrow \mathsf{pad}_{128}(M_{\ell_M-1})$
        $\mathsf{Const}_M \leftarrow 2 \oplus (1 \ll 2)$
    else
        $\mathsf{Const}_M \leftarrow 3 \oplus (1 \ll 2)$
    end if
end if
 // State initialization
$S_L \| S_R \leftarrow \textsc{Sparkle256}_{10}\big(N \| K\big)$ with $|S_L| = 128$ and $|S_R| = 128$
 // Processing of associated data
if $A \neq \epsilon$ then
    for all $j = 0, \dots, \ell_A - 2$ do
        $S_L \| S_R \leftarrow \textsc{Sparkle256}_7\big((\rho_1(S_L, A_j) \oplus S_R) \| S_R\big)$
    end for
     // Finalization if message is empty
    $S_L \| S_R \leftarrow \textsc{Sparkle256}_{10}\big((\rho_1(S_L, A_{\ell_A-1}) \oplus S_R \oplus \mathsf{Const}_A) \| (S_R \oplus \mathsf{Const}_A)\big)$
end if
 // Encrypting
if $M \neq \epsilon$ then
    for all $j = 0, \dots, \ell_M - 2$ do
        $C_j \leftarrow \rho_2(S_L, M_j)$
        $S_L \| S_R \leftarrow \textsc{Sparkle256}_7\big((\rho_1(S_L, M_j) \oplus S_R) \| S_R\big)$
    end for
    $C_{\ell_M-1} \leftarrow \mathsf{trunc}_t\big(\rho_2(S_L, M_{\ell_M-1})\big)$
     // Finalization
    $S_L \| S_R \leftarrow \textsc{Sparkle256}_{10}\big((\rho_1(S_L, M_{\ell_M-1}) \oplus S_R \oplus \mathsf{Const}_M) \| (S_R \oplus \mathsf{Const}_M)\big)$
end if

return $(C_0 \| C_1 \| \dots \| C_{\ell_M-1}, S_R \oplus K)$
```

Algorithm 18 $\textsc{Schwaemm128-128-Dec}$

Input: (K, N, A, C, T) where $K \in \mathbb{F}_2^{128}$ is a key, $N \in \mathbb{F}_2^{128}$ is a nonce, $A, C \in \mathbb{F}_2^*$ and $T \in \mathbb{F}_2^{128}$

Output: Decryption M of C if the tag T is valid, $\perp$ otherwise

```
if $A \neq \epsilon$ then
    $A_0 \| A_1 \| \ldots \| A_{\ell_A - 1} \leftarrow A$ with $\forall i \in \{0, \ldots, \ell_A - 2\} : |A_i| = 128$ and $1 \leq |A_{\ell_A-1}| \leq 128$
    if $|A_{\ell_A-1}| < 128$ then
        $A_{\ell_A-1} \leftarrow \mathsf{pad}_{128}(A_{\ell_A-1})$
        $\mathsf{Const}_A \leftarrow 0 \oplus (1 \ll 2)$
    else
        $\mathsf{Const}_A \leftarrow 1 \oplus (1 \ll 2)$
    end if
end if
if $C \neq \epsilon$ then
    $C_0 \| C_1 \| \ldots \| C_{\ell_M - 1} \leftarrow C$ with $\forall i \in \{0, \ldots, \ell_M - 2\} : |C_i| = 128$ and $1 \leq |C_{\ell_M-1}| \leq 128$
    $t \leftarrow |C_{\ell_M-1}|$
    if $|C_{\ell_M-1}| < 128$ then
        $C_{\ell_M-1} \leftarrow \mathsf{pad}_{128}(C_{\ell_M-1})$
        $\mathsf{Const}_M \leftarrow 2 \oplus (1 \ll 2)$
    else
        $\mathsf{Const}_M \leftarrow 3 \oplus (1 \ll 2)$
    end if
end if
// State initialization
$S_L \| S_R \leftarrow \textsc{Sparkle256}_{10}(N \| K)$ with $|S_L| = 128$ and $|S_R| = 128$
// Processing of associated data
if $A \neq \epsilon$ then
    for all $j = 0, \ldots, \ell_A - 2$ do
        $S_L \| S_R \leftarrow \textsc{Sparkle256}_7\big((\rho_1(S_L, A_j) \oplus S_R) \| S_R\big)$
    end for
    // Finalization if ciphertext is empty
    $S_L \| S_R \leftarrow \textsc{Sparkle256}_{10}\big((\rho_1(S_L, A_{\ell_A-1}) \oplus S_R \oplus \mathsf{Const}_A) \| (S_R \oplus \mathsf{Const}_A)\big)$
end if
// Decrypting
if $C \neq \epsilon$ then
    for all $j = 0, \ldots, \ell_M - 2$ do
        $M_j \leftarrow \rho_2'(S_L, C_j)$
        $S_L \| S_R \leftarrow \textsc{Sparkle256}_7\big((\rho_1'(S_L, C_j) \oplus S_R) \| S_R\big)$
    end for
    $M_{\ell_M-1} \leftarrow \mathsf{trunc}_t\big(\rho_2'(S_L, C_{\ell_M-1})\big)$
    // Finalization and tag verification
    if $t < 128$ then
        $S_L \| S_R \leftarrow \textsc{Sparkle256}_{10}\big((\rho_1(S_L, \mathsf{pad}_{128}(M_{\ell_M-1})) \oplus S_R \oplus \mathsf{Const}_M) \| (S_R \oplus \mathsf{Const}_M)\big)$
    else
        $S_L \| S_R \leftarrow \textsc{Sparkle256}_{10}\big((\rho_1'(S_L, C_{\ell_M-1}) \oplus S_R \oplus \mathsf{Const}_M) \| (S_R \oplus \mathsf{Const}_M)\big)$
    end if
end if
if $S_R \oplus K = T$ then
    return $(M_0 \| M_1 \| \ldots \| M_{\ell_M-1})$
else
    return $\perp$
end if
```

Algorithm 19 SCHWAEMM256-256-ENC

Input: (K, N, A, M) where $K \in \mathbb{F}_2^{256}$ is a key, $N \in \mathbb{F}_2^{256}$ is a nonce and $A, M \in \mathbb{F}_2^*$

Output: (C, T), where $C \in \mathbb{F}_2^*$ is the ciphertext and $T \in \mathbb{F}_2^{256}$ is the authentication tag

```
 // Padding the associated data and message
if $A \neq \epsilon$ then
    $A_0 \| A_1 \| \dots \| A_{\ell_A - 1} \leftarrow A$ with $\forall i \in \{0, \dots, \ell_A - 2\} : |A_i| = 256$ and $1 \leq |A_{\ell_A - 1}| \leq 256$
    if $|A_{\ell_A - 1}| < 256$ then
        $A_{\ell_A - 1} \leftarrow \mathsf{pad}_{256}(A_{\ell_A - 1})$
        $\mathsf{Const}_A \leftarrow 0 \oplus (1 \ll 4)$
    else
        $\mathsf{Const}_A \leftarrow 1 \oplus (1 \ll 4)$
    end if
end if
if $M \neq \epsilon$ then
    $M_0 \| M_1 \| \dots \| M_{\ell_M - 1} \leftarrow M$ with $\forall i \in \{0, \dots, \ell_M - 2\} : |M_i| = 256$ and $1 \leq |M_{\ell_M - 1}| \leq 256$
    $t \leftarrow |M_{\ell_M - 1}|$
    if $|M_{\ell_M - 1}| < 256$ then
        $M_{\ell_M - 1} \leftarrow \mathsf{pad}_{256}(M_{\ell_M - 1})$
        $\mathsf{Const}_M \leftarrow 2 \oplus (1 \ll 4)$
    else
        $\mathsf{Const}_M \leftarrow 3 \oplus (1 \ll 4)$
    end if
end if
 // State initialization
$S_L \| S_R \leftarrow \textsc{Sparkle512}_{12}\big(N \| K\big)$ with $|S_L| = 256$ and $|S_R| = 256$
 // Processing of associated data
if $A \neq \epsilon$ then
    for all $j = 0, \dots, \ell_A - 2$ do
        $S_L \| S_R \leftarrow \textsc{Sparkle512}_8\big((\rho_1(S_L, A_j) \oplus S_R) \| S_R\big)$
    end for
     // Finalization if message is empty
    $S_L \| S_R \leftarrow \textsc{Sparkle512}_{12}\big((\rho_1(S_L, A_{\ell_A - 1}) \oplus S_R \oplus \mathsf{Const}_A) \| (S_R \oplus \mathsf{Const}_A)\big)$
end if
 // Encrypting
if $M \neq \epsilon$ then
    for all $j = 0, \dots, \ell_M - 2$ do
        $C_j \leftarrow \rho_2(S_L, M_j)$
        $S_L \| S_R \leftarrow \textsc{Sparkle512}_8\big((\rho_1(S_L, M_j) \oplus S_R) \| S_R\big)$
    end for
    $C_{\ell_M - 1} \leftarrow \mathsf{trunc}_t\big(\rho_2(S_L, M_{\ell_M - 1})\big)$
     // Finalization
    $S_L \| S_R \leftarrow \textsc{Sparkle512}_{12}\big((\rho_1(S_L, M_{\ell_M - 1}) \oplus S_R \oplus \mathsf{Const}_M) \| (S_R \oplus \mathsf{Const}_M)\big)$
end if

return $(C_0 \| C_1 \| \dots \| C_{\ell_M - 1}, S_R \oplus K)$
```

Algorithm 20 SCHWAEMM256-256-DEC
Input: (K, N, A, C, T) where $K \in \mathbb{F}_2^{256}$ is a key, $N \in \mathbb{F}_2^{256}$ is a nonce, $A, C \in \mathbb{F}_2^*$ and $T \in \mathbb{F}_2^{256}$
Output: Decryption M of C if the tag T is valid, $\perp$ otherwise

```
if $A \neq \epsilon$ then
    $A_0 \| A_1 \| \dots \| A_{\ell_A - 1} \leftarrow A$ with $\forall i \in \{0, \dots, \ell_A - 2\} : |A_i| = 256$ and $1 \leq |A_{\ell_A-1}| \leq 256$
    if $|A_{\ell_A-1}| < 256$ then
        $A_{\ell_A-1} \leftarrow \mathsf{pad}_{256}(A_{\ell_A-1})$
        $\mathsf{Const}_A \leftarrow 0 \oplus (1 \ll 4)$
    else
        $\mathsf{Const}_A \leftarrow 1 \oplus (1 \ll 4)$
    end if
end if
if $C \neq \epsilon$ then
    $C_0 \| C_1 \| \dots \| C_{\ell_M - 1} \leftarrow C$ with $\forall i \in \{0, \dots, \ell_M - 2\} : |C_i| = 256$ and $1 \leq |C_{\ell_M-1}| \leq 256$
    $t \leftarrow |C_{\ell_M-1}|$
    if $|C_{\ell_M-1}| < 256$ then
        $C_{\ell_M-1} \leftarrow \mathsf{pad}_{256}(C_{\ell_M-1})$
        $\mathsf{Const}_M \leftarrow 2 \oplus (1 \ll 4)$
    else
        $\mathsf{Const}_M \leftarrow 3 \oplus (1 \ll 4)$
    end if
end if
// State initialization
$S_L \| S_R \leftarrow \textsc{Sparkle512}_{12}\big(N \| K\big)$ with $|S_L| = 256$ and $|S_R| = 256$
// Processing of associated data
if $A \neq \epsilon$ then
    for all $j = 0, \dots, \ell_A - 2$ do
        $S_L \| S_R \leftarrow \textsc{Sparkle512}_8\big((\rho_1(S_L, A_j) \oplus S_R) \| S_R\big)$
    end for
    // Finalization if ciphertext is empty
    $S_L \| S_R \leftarrow \textsc{Sparkle512}_{12}\big((\rho_1(S_L, A_{\ell_A-1}) \oplus S_R \oplus \mathsf{Const}_A) \| (S_R \oplus \mathsf{Const}_A)\big)$
end if
// Decrypting
if $C \neq \epsilon$ then
    for all $j = 0, \dots, \ell_M - 2$ do
        $M_j \leftarrow \rho'_2(S_L, C_j)$
        $S_L \| S_R \leftarrow \textsc{Sparkle512}_8\big((\rho'_1(S_L, C_j) \oplus S_R) \| S_R\big)$
    end for
    $M_{\ell_M-1} \leftarrow \mathsf{trunc}_t\big(\rho'_2(S_L, C_{\ell_M-1})\big)$
    // Finalization and tag verification
    if $t < 256$ then
        $S_L \| S_R \leftarrow \textsc{Sparkle512}_{12}\big((\rho_1(S_L, \mathsf{pad}_{256}(M_{\ell_M-1})) \oplus S_R \oplus \mathsf{Const}_M) \| (S_R \oplus \mathsf{Const}_M)\big)$
    else
        $S_L \| S_R \leftarrow \textsc{Sparkle512}_{12}\big((\rho'_1(S_L, C_{\ell_M-1}) \oplus S_R \oplus \mathsf{Const}_M) \| (S_R \oplus \mathsf{Const}_M)\big)$
    end if
end if
if $S_R \oplus K = T$ then
    return $(M_0 \| M_1 \| \dots \| M_{\ell_M-1})$
else
    return $\perp$
end if
```

4.5.6 Recommendations for instances

We recommend the joint evaluation of the AEAD and hashing schemes that use the same Sparkle version as the underlying permutation. The particular pairings are shown in Table 4.4. Note that we do not pair Schwaemm128-128 with a hashing algorithm as we did not specify a member of the Esch family that employs Sparkle256.

Hashing	AEAD	based on
Esch256	Schwaemm256-128	Sparkle384
Esch256	Schwaemm192-192	Sparkle384
Esch384	Schwaemm256-256	Sparkle512
–	Schwaemm128-128	Sparkle256

Table 4.4: Recommendations for joint evaluation of Esch and Schwaemm. The first pairing refers to the primary member of both functionalities.

4.6 Implementation Aspects

This section presents some characteristics of Sparkle, with focus on software implementations.

Alzette

The ARX-box Alzette is an important part of Sparkle, and as such, was designed to provide good security bounds, but also efficient implementation. The rotation amounts have been carefully chosen to be a multiple of eight bits or one bit from it. On 8 or 16 bit architectures these rotations can be efficiently implemented using move, swap, and 1-bit rotate instructions. On ARM processors, operations of the form $z \leftarrow x <op> (y \lll n)$ can be executed with a single instruction in a single clock cycle, irrespective of the rotation distance.

Alzette itself operates over two 32-bit words of data, with an extra 32-bit constant value. This allows the full computation to happen in-register in AVR, MSP and ARM architectures, whereby the latter is able to hold at least 4 Alzette instances entirely in registers. This in turn reduces load-store overheads and contributes to the performance of the permutation.

The consistency of operations across branches, which means that each branch executes the same sequence of instructions, allows one to either focus on small code size (by implementing the Alzette layer in a loop), or on architectures with more registers, execute two or more branches to exploit instruction pipelining.

This consistency of operations also allows some degree of parallelism, namely by using Single Instruction Multiple Data (SIMD) instructions. SIMD is a type of computational model that executes the same operation on multiple operands. The branch structure of Sparkle makes it possible to manipulate the state through SIMD instructions. In addition, the small size of the state also allows it to fit in most popular SIMD engines, such as ARM's NEON and Intel's SSE or AVX. Due to the layout of Alzette a SIMD implementation can be created by packing $x_0 \dots x_{n_b}$, $y_0 \dots y_{n_b}$, and $c_0 \dots c_{n_b}$ each in a vector register. That allows 128-bit SIMD architectures such as NEON to execute four Alzette instances in parallel, or even eight instances when using x86 AVX2 instructions.

Linear Layer

It is, of course, possible to implement the branch permutation at the end of the linear layer like a branch rotation in the right half followed by a swap of the left and right branches. However, the combination of the two operation has a unique cycle meaning that it can be implemented simply in one loop, as shown in Algorithm 21. It is the strategy we used in Section 4.7. Note that it is not necessary to reduce the indices modulo w or $w/2$, which greatly simplifies this implementation of the linear layer.

Algorithm 21 The permutation of w branch used in $\mathcal{L}_w$
Input/Output: $(Z_0, ..., Z_{w-1}) \in (\mathbb{F}_2^{64})^w$

```
$Z' \leftarrow Z_0$
for all $i \in \{1, ..., w/2 - 1\}$ do
    $Z_{i-1} \leftarrow Z_{i+w/2}$
    $Z_{i+w/2} \leftarrow Z_i$
end for
$Z_{w/2} = Z'$
return $(Z_0, ..., Z_{w-1})$
```

On an optimized implementation, the linear layer's branch permutations can be abstracted on an unrolled implementation, at the cost of code size.

Parameterized implementations

Parameterized implementations, offering support to all instances of the algorithm, are easily done and contribute to a small code size. It also facilitates the writing of macro-based code that compiles binaries for a specific instance. An implementation of Sparkle can be parameterized by the number of rounds and branches. Schwaemm implementations need only the rate, capacity, and round numbers. Similarly, Esch needs only the number of branches and steps. Beyond that, a single implementation of Sparkle is sufficient for all instances of Schwaemm and Esch, making optimization, implementation, and testing easier.

4.6.1 Hardware Implementation

Both Esch and Schwaemm are based on the Sparkle permutations, where addition, rotation, and XOR are the main components. There exist a number of different design approaches for a 32-bit adder as the largest component in hardware. The simplest variant is a conventional Ripple-Carry Adder (RCA) composed of 32 Full Adder (FA) cells. RCAs are very efficient in terms of area requirements, but their delay increases linearly with the bit-length of the adder. Alternatively, if an implementation requires a short critical path, the adder can also take the form of a Carry-Lookahead Adder (CLA), Carry-Skip Adder (CSA), or KoggeStone Adder (KSA), which have a delay that grows logarithmically with the word size at the cost of higher area overhead. Rotations are free in hardware as they are just a simple wiring, and the implementation of XOR operation is pretty straightforward.

To achieve a high-throughput implementation, each round of a Sparkle permutation can be implemented as a fully combinatorial circuit, performed by a single clock cycle.

In this approach, the *ARX-box* Alzette is instantiated multiple times depending on the number of branches, i.e., n_b times in parallel followed by an instance of linear layer $\mathcal{L}_{n_b}$ to realize a round function of SPARKLE permutation. To reduce the area overhead, the *ARX-box* Alzette can be instantiated once and re-used multiple times to perform the round function. Hence, each round can be performed in n_b clock cycles, leading to higher latency but lower area overhead. Moreover, the design can be optimized further for small size of silicon area. Since only four different amount of rotations - namely 16, 17, 24, and 31 bits - are used, it can be simply implemented by 32 instances of a 4-to-1 multiplexer. Hence, a minimalist hardware designer can realize the *ARX-box* Alzette by a 32-bit adder, a 32-bit XOR, a 32-bit wide 4-to-1 multiplexer, and a control unit. Following this approach, each round of SPARKLE permutation can be executed in $4n_b$ clock cycles provided that an instance of linear layer $\mathcal{L}_{n_b}$ is implemented in the design.

4.6.2 Protection against Side-Channel Attacks

A straightforward implementation of a symmetric cryptographic algorithm such as SCHWAEMM is normally vulnerable to side-channel attacks, in particular to Differential Power Anaylsis (DPA). Timing attacks and conventional Simple Power Analysis (SPA) attacks are a lesser concern since the specification of SCHWAEMM does not contain any conditional statement (e.g. if-then-else clauses) that depend on secret data. A well-known and widely-used countermeasure against DPA attacks is *masking*, which can be realized in both hardware and software. Masking aims to conceal every key-dependent variable with a random value called mask (or a set of masks for high orders) to decorrelate the sensitive data of the algorithm from the data that is actually processed on the device. The basic principle is related to the idea of secret sharing because every sensitive variable is split up into $n \geq 2$ “shares” so that any combination of up to $d = n - 1$ shares is statistically independent of any secret value. These n shares have to be processed separately during the execution of the algorithm (to ensure their leakages are independent of each other) and then recombined at the end to yield the correct result.

Depending on the actual operation to be protected against DPA, a masking scheme can be Boolean (using logical XOR), arithmetic (using modular addition or modular subtraction) or multiplicative (using modular multiplication). Since SCHWAEMM is an ARX design and, consequently, involves arithmetic and Boolean operations, the masks have to be converted from one form to the other without introducing any kind of leakage. There exists an abundant literature on mask conversion techniques and it is nowadays well understood how one can convert efficiently from arithmetic masks to Boolean masks and vice versa, see e.g. [CGV14]. An alternative approach is to compute the arithmetic operations (i.e. modular addition) directly on Boolean shares as described in e.g. [Cor+15; SMG15]. In summary, SCHWAEMM profits from the vast body of research on masking schemes for ARX designs and can be effectively and efficiently protected against DPA attacks.

4.6.3 Implementation Results

Accompanying this submission are reference and optimized C implementations of different instances of SCHWAEMM and ESCH, as well as assembler implementations of the SPARKLE permutation for 8-bit AVR ATmega and 32-bit ARM Cortex-M microcontrollers. The AVR assembler code for SPARKLE is parameterized by the number of branches and the number of steps, and complies with the interface of the optimized

C implementation. Therefore, the assembler implementation can serve as a “plug-in” replacement for the optimized C code to further increase the performance on AVR devices. Thanks to the parameterization, the assembler implementation of SPARKLE provides the full functionality needed by the different instances of SCHWAEMM and ESCH.

In contrast to AVR, we developed separate assembler implementations for SPARKLE256, SPARKLE384, and SPARKLE512 for ARM, which are “branch-unrolled” in the sense that the number of branches is hard-coded and not passed as argument anymore. However, all three ARM assembler implementations are still parameterized by the number of steps so that a unique assembler function is capable to support both the slim and big number of steps specified in Table 4.1. The main reason why it makes sense to develop three branch-unrolled Assembler implementations of SPARKLE for ARM but not for AVR is the large register space of the former architecture, which is capable to accommodate the full state of SPARKLE256 and SPARKLE384, thereby significantly reducing the number of load/store operations. Unfortunately, this approach for optimizing the two smaller SPARKLE instances can not be applied in a single branch-parameterized assembler function. It is nonetheless possible to have a “plug-in” assembler replacement for the fully-parameterized C implementation of the SPARKLE permutation by writing a wrapper over the three SPARKLE functions that has the same interface as the C implementation (i.e. this wrapper is parameterized by both the number of steps and the number of branches). The wrapper simple checks the number of branches and then calls the corresponding variant of the assembler function, i.e. SPARKLE256 when the number of branches is 4, SPARKLE384 when the number of branches is 6, and SPARKLE512 when the number of branches is 8.

The execution times and throughputs of our assembler implementations of the SPARKLE permutation for AVR and ARM are summarized in Table 4.5. On AVR, the assembler code is approximately four times faster than the optimized C code (compiled with `avr-gcc 5.4.0`), which is roughly in line with the results observed in [CDG19]. The main reasons for the relatively bad performance of the compiled code are a poor register allocation strategy (which causes many unnecessary memory accesses) and the non-optimal code generated for the rotations compared to hand-optimized assembler code. Our AVR assembler implementation is also relatively small in terms of code size (702 bytes) and occupies only 21 bytes on the stack (for callee-saved registers). All execution times for AVR were determined with help of the cycle-accurate instruction set simulator of Atmel Studio 7 using the ATmega128 microcontroller as target device.

The performance gap between the compiled C code and the hand-written assembler code is a bit smaller on ARM, namely by a factor of roughly 2.5 when executed on a Cortex-M3. However, it has to be taken into account that the assembler functions are “branch-unrolled,” whereas the C version is fully parameterized. The C implementation was compiled with Keil MicroVision v5.24.2.0 using optimization level `-O2`. Obviously, the large register space and the “free” rotations of the ARM architecture make it easier for a compiler to generate efficient code. The binary code size of the assembler implementations of SPARKLE for ARM ranges between 348 and 628 bytes and they occupy at most 52 bytes on the stack, of which 36 bytes are due to callee-saved registers (see Table 4.6). All execution times for ARM specified in Table 4.5 were obtained with the cycle-accurate instruction set simulator of Keil MicroVision

Permutation	Rounds	AVR		ARM	
		C	asm	C	asm
Sparkle256	7 (slim)	697 (22305)	179 (5728)	46 (1487)	19 (605)
	10 (big)	992 (31761)	254 (8146)	66 (2111)	26 (842)
Sparkle384	7 (slim)	680 (32679)	173 (8318)	45 (2173)	19 (930)
	11 (big)	1066 (51215)	271 (13022)	71 (3397)	30 (1430)
Sparkle512	8 (slim)	768 (49169)	194 (12454)	51 (3263)	23 (1489)
	12 (big)	1150 (73633)	291 (18638)	76 (4879)	35 (2209)

Table 4.5: Performance of the Sparkle permutation on an 8-bit AVR ATmega128 and a 32-bit ARM Cortex-M3 microcontroller. The results are given in cycles/byte, with the number inside parentheses representing the total cycle count for an execution of the permutation.

using a generic Cortex-M3 model as target device.[6] It should be noted that the results for ARM are based on assembler implementations that were optimized to achieve a balance between (binary) code size and speed, which means we refrained from certain optimization techniques like full loop unrolling (i.e. unrolling not only the branches but also the steps). However, we also developed more aggressively speed-optimized versions of the three permutations where we fully unrolled the step-loop, which reduces the execution time by between 15% and 18% (e.g. 149 cycles in the case of Sparkle384 with the slim number of steps). This performance gain is not only due to the elimination of the overhead of the step-loop. Indeed, the execution time of the linear layer could be further reduced: concretely, the 1-branch left-rotation of the right-side branches in the linear layer is done "implicitly". The downside of this full loop unrolling is a massive increase in code size (e.g. by a factor of almost 6 for the slim version of Sparkle384).

Permutation	Code Size (byte)	Stack Usage (byte)
Sparkle256	316+32	40
Sparkle384	452+32	48
Sparkle512	596+32	52

Table 4.6: Code size and stack consumption of the Sparkle permutations on a 32-bit ARM Cortex-M3 microcontroller. The code size is given as the number of bytes the permutation occupies in the `text` segment plus the 32 bytes for the round constants.

Besides Sparkle, a multitude of other permutation-based designs was submitted to the NIST lightweight cryptography standardization process. Three of those designs, namely Ascon, Gimli, and Xoodoo, come with optimized (i.e. fully unrolled) assembly implementations of the underlying permutation for the Cortex-M series of ARM

[6]As mentioned on `http://www2.keil.com/mdk5/simulation`, the Keil simulator assumes ideal conditions for memory accesses and does not simulate wait states for data or code fetches. Therefore, the timings in Table 4.5 should be seen as lower bounds of the actual execution times one will get on a real Cortex-M3 device. The fact that the Keil simulator does not take flash wait-states into account may also explain why our simulated execution time for the Gimli permutation (1041 cycles) differs slightly from the 1047 cycles specified in Section 5.5 of [Ber+17b].

Permutation	Code Size (byte)	Time (cycles)	Time/Rate (cycles/byte)
ASCON (8 rounds)	1810	499	31.19
GIMLI (24 rounds)	3950	1041	65.06
SPARKLE384 (7 steps)	2820	781	24.40
XOODOO (12 rounds)	2376	657	27.38

TABLE 4.7: Comparison of fully unrolled ARMv7-M Assembler implementations of the permutations of ASCON, SPARKLE384, GIMLI and XOODOO on a Cortex-M3 microcontroller.

microcontrollers. Table 4.7 compares the execution time and code size of the permutations of ASCON, GIMLI and XOODOO with a fully-unrolled version of SPARKLE384, which is the permutation used by the primary instance of SCHWAEMM and ESCH.[7] As mentioned before, full loop unrolling reduces the execution time of SPARKLE384 from 930 to 781 clock cycles, but this reduction by 149 cycles comes at the expense of an almost 6-fold increase of code size. Also given in Table 4.7 is the throughput (in cycles per byte) of the permutations, which is simply the execution time of the permutation divided by the rate of the main instance of the corresponding AEAD algorithm (16 bytes for ASCON and GIMLI, 32 bytes for SCHWAEMM256-128, and 24 bytes for Xoodyak). SPARKLE384 achieves the highest throughput, closely followed by XOODOO and ASCON. GIMLI reaches less than half of the throughput of the other three permutations, but it has to be taken into account that the GIMLI AEAD algorithm aims for 256 bits of security.

Instance	64 bytes of data		1536 bytes of data	
	Pure C	C + asm	Pure C	C + asm
SCHWAEMM128-128	2444 (156416)	712 (45583)	1421 (2182899)	387 (594898)
SCHWAEMM256-128	2105 (134748)	596 (38166)	1071 (1644606)	302 (464347)
SCHWAEMM192-192	2594 (165994)	727 (46526)	1399 (2148858)	395 (606716)
SCHWAEMM256-256	3014 (192918)	839 (53704)	1574 (2417064)	434 (666554)
ESCH256	2714 (173678)	893 (57187)	1978 (3038834)	559 (860071)
ESCH384	4732 (302837)	1308 (83725)	2992 (4595649)	830 (161717)

TABLE 4.8: Benchmarking results for the different instances of SCHWAEMM and ESCH on an AVR ATmega128 microcontroller when processing 64 and 1536 bytes of data, respectively (in the case of SCHWAEMM the benchmarked operation is encryption and the length of the associated data is 0). The results are given in cycles/byte, with the number inside parentheses representing the total cycle count for processing the specified amount of data.

Table 4.8 shows the AVR execution times and throughputs of the different instances of SCHWAEMM and ESCH instances when processing a small amount (64 bytes) and a large amount (1536 bytes) of data, respectively. As before, all execution times were obtained with the cycle-accurate simulator of Atmel Studio 7 using an ATmega128 as target device. The results in the "C + asm" columns refer to a C implementation

[7]We took the ARM Assembler source code of GIMLI from `http://gimli.cr.yp.to/gimli-20170627.tar.gz` and converted it from the GNU syntax to the Keil syntax. The source code of XOODOO contained in the eXtended Keccak Code Package (XKCP) at `http://github.com/XKCP/XKCP/tree/master/lib/low/Xoodoo` was already in Keil syntax.

Instance	64 bytes of data		1536 bytes of data	
	Pure C	C + asm	Pure C	C + asm
SCHWAEMM128-128	148 (9491)	69 (4384)	101 (155495)	46 (70440)
SCHWAEMM256-128	154 (9851)	74 (4715)	77 (118917)	37 (57109)
SCHWAEMM192-192	189 (12066)	89 (5698)	100 (153597)	47 (72077)
SCHWAEMM256-256	219 (14029)	111 (7072)	113 (173051)	56 (86284)
ESCH256	198 (12654)	90 (5774)	114 (221678)	66 (101454)
ESCH384	341 (21847)	165 (10561)	216 (332623)	105 (161717)

TABLE 4.9: Benchmarking results for the different instances of SCHWAEMM and ESCH on an ARM Cortex-M3 microcontroller when processing 64 and 1536 bytes of data, respectively (in the case of SCHWAEMM the benchmarked operation is encryption and the length of the associated data is 0). The results are given in cycles/byte, with the number inside parentheses representing the total cycle count for processing the specified amount of data.

that uses the hand-written assembler code for the SPARKLE permutation. Table 4.9 summarizes the corresponding results for an ARM Cortex-M3 device.

In order to compare the performance of ESCH256 (using the assembler implementation of SPARKLE as sub-function) with that of other (lightweight) hash functions, we simulated the time it needs to hash a 500-byte message on an 8-bit AVR ATmega128 microcontroller. Indeed, determining the execution time required for hashing a 500-byte message on AVR is a well-established way to generate benchmarks for a comparison of lightweight hash functions. According to our simulation results, the mixed C and assembler implementation of ESCH256 has an execution time of 289131 clock cycles, which translates to a hash rate of 578 cycles/byte. The binary code size of ESCH256 is 1428 bytes. Table 4.10 summarizes the implementation results of ESCH256, SHA-2, SHA-3, some SHA-3 finalists, as well as GIMLI [Ber+17b]. Our hash rate of 578 cycles/byte for ESCH256 compares favorably with the results of the SHA-3 finalists and is beaten only by BLAKE-256 and SHA-256. However, it must be taken into account that the results reported in [Bal+13] were obtained with "pure" assembler implementations, whereas ESCH256 contains hand-optimized assembler code only for the SPARKLE permutation. We expect that a fully-optimized implementation of ESCH256 with all its components written in assembler has the potential to be faster BLAKE-256 and get very close to (or even outperform) SHA-256.

A comparison of the performance of hash functions is easily possible because there exist a number of implementation results in the literature (e.g. [Bal+13]) that were obtained in a consistent fashion, in particular by measuring the execution time required for hashing a 500-byte message on AVR. Unfortunately, there seems to be no similarly established way of generating benchmarking results for lightweight authenticated encryption algorithms since the results one can find in the literature were obtained with completely different lengths of plaintexts/ciphertexts and associated data.

Hash function	Ref.	Throughput (c/b)	Code size (b)
ESCH256	This paper	578	1428
BLAKE-256	[Bal+13]	562	1166
GIMLI-Hash small	[Ber+a]	1610	778*
GIMLI-Hash fast	[Ber+a]	725	19218*
GROESTL-256	[Bal+13]	686	1400
JH-256	[Bal+13]	5062	1020
KECCAK†	[Bal+13]	1432	868
SHA-256	[Bal+13]	532	1090

* The code size corresponds to the permutation alone.

† The version of KECCAK considered is KECCAK$[r = 1088, c = 512]$.

TABLE 4.10: Comparison of ESCH256 with other hash functions producing a 256-bit digest. The number of cycles and the throughput were obtained by hashing a 500-byte message on an AVR microcontroller. The implementation of ESCH256 contains hand-optimized assembler code only for the permutation, whereas the implementations of all other hash functions were written entirely in assembler.

4.7 C Implementation of Sparkle

All permutations in the SPARKLE family are implemented by the following function, where `nb` is the number of branches (4 for SPARKLE256, 6 for SPARKLE384 and 8 for SPARKLE512) and where `ns` is the number of steps.

```c
#define MAX_BRANCHES 8
#define ROT(x, n) (((x) >> (n)) | ((x) << (32-(n))))
#define ELL(x) (ROT(((x) ^ ((x) << 16)), 16))

// Round constants
static const uint32_t RCON[MAX_BRANCHES] = {       \
  0xB7E15162, 0xBF715880, 0x38B4DA56, 0x324E7738, \
  0xBB1185EB, 0x4F7C7B57, 0xCFBFA1C8, 0xC2B3293D  \
};

void sparkle(uint32_t *state, int nb, int ns)
{
  int i, j;  // Step and branch counter
  uint32_t rc, tmpx, tmpy, x0, y0;

  for(i = 0; i < ns; i ++) {
    // Counter addition
    state[1] ^= RCON[i%MAX_BRANCHES];
    state[3] ^= i;
    // ARXBox layer
    for(j = 0; j < 2*nb; j += 2) {
      rc = RCON[j>>1];
      state[j] += ROT(state[j+1], 31);
      state[j+1] ^= ROT(state[j], 24);
      state[j] ^= rc;
      state[j] += ROT(state[j+1], 17);
      state[j+1] ^= ROT(state[j], 17);
      state[j] ^= rc;
      state[j] += state[j+1];
      state[j+1] ^= ROT(state[j], 31);
      state[j] ^= rc;
      state[j] += ROT(state[j+1], 24);
      state[j+1] ^= ROT(state[j], 16);
      state[j] ^= rc;
    }
    // Linear layer
    tmpx = x0 = state[0];
    tmpy = y0 = state[1];
    for(j = 2; j < nb; j += 2) {
      tmpx ^= state[j];
      tmpy ^= state[j+1];
    }
    tmpx = ELL(tmpx);
    tmpy = ELL(tmpy);
    for (j = 2; j < nb; j += 2) {
      state[j-2] = state[j+nb] ^ state[j] ^ tmpy;
      state[j+nb] = state[j];
      state[j-1] = state[j+nb+1] ^ state[j+1] ^ tmpx;
      state[j+nb+1] = state[j+1];
    }
    state[nb-2] = state[nb] ^ x0 ^ tmpy;
    state[nb] = x0;
    state[nb-1] = state[nb+1] ^ y0 ^ tmpx;
    state[nb+1] = y0;
  }
}
```

Part IV

Cryptanalysis of ARX ciphers

Chapter 5

Meet in the Filter

This chapter is based on the paper titled "Meet-in-the-filter and dynamic counting with applications do Speck", which was submitted for review to Asiacrypt 2022 [Ale+22].

My main contribution on this chapter was the base design of the attack based on the clustering characteristic of the trails, the trail enumeration and search procedures, as well as the necessary implementations and tools, focused primarily on Speck64, with no specific optimization for Speck32. The key recovery procedure, special trail filtering for Speck32, and more accurate complexity analysis was executed by the other authors of the paper. The full paper is available in https://eprint.iacr.org/2022/673/

In this paper, we propose a new cryptanalytic tool for differential cryptanalysis, called *meet-in-the-filter* (MiF). It is suitable for ciphers with a slow or incomplete diffusion layer such as the ones based on Addition-Rotation-XOR (ARX). The main idea of the MiF technique is to stop the difference propagation earlier in the cipher, allowing to use differentials with higher probability. This comes at the expense of a deeper analysis phase in the bottom rounds possible due to the slow diffusion of the target cipher. The MiF technique uses a meet-in-the-middle matching to construct differential trails connecting the differential's output and the ciphertext difference. The proposed trails are used in the key recovery procedure, reducing time complexity and allowing flexible time-data trade-offs. In addition, we show how to combine MiF with a *dynamic counting* technique for key recovery. We illustrate this in practice by reporting improved attacks on the ARX-based family of block ciphers Speck. We improve the time complexities of the best known attacks up to 15 rounds of Speck32 and 20 rounds of Speck64/128. Notably, our new attack on 11 rounds of Speck32 has practical analysis and data complexities of $2^{24.66}$ and $2^{26.70}$ respectively, and was experimentally verified, recovering the master key in a matter of seconds. It significantly improves the previous deep learning-based attack by Gohr from CRYPTO 2019, which has time complexity 2^{38}.

5.1 Introduction

Differential cryptanalysis (DC) is one of the most powerful techniques for analyzing symmetric-key cryptographic algorithms. It has been proposed by Biham and Shamir in 1991 [BS91] and since then has been used to successfully attack numerous symmetric-key primitives, including ciphers, hash functions, and MACs. Nowadays, resistance to DC is one of the basic properties that a symmetric-key algorithm must satisfy and new cryptographic designs often come with proofs of such resistance.

In DC, the attacker traces the propagation of differences (most commonly expressed in terms of the XOR operation) between plaintexts through multiple rounds of the analyzed primitive. By analyzing differences rather than plaintexts, the attacker effectively cancels out the action of the unknown round keys (also typically mixed in by an XOR). In this way, a trace of differences over multiple rounds can be computed, which is called a *differential trail.* The latter typically holds with certain probability $p > 2^{-n}$ for an n-bit state and acts as a *distinguisher* of the analyzed cipher from a random permutation.

A typical DC attack starts with the derivation of a distinguisher on r rounds with probability p. It is then used to attack $r + u$ rounds of the cipher, where u is some number of rounds added after the distinguisher. In the attack, the attacker guesses (at least partially) the last u round keys in order to invert the last u rounds and to compute the output difference after r rounds. If this difference matches the output difference of the distinguisher, then, with some probability, the guess for the last round keys must have been correct. Extra l rounds are often also added at the top of the distinguisher resulting in an attack on $l+r+u$ rounds. DC is a statistical attack, meaning that the described process has to be repeated for many (at least p^{-1}) chosen plaintexts with a given input difference in order to successfully recover the last round keys with a sufficient success probability. Over the years there have been multiple extensions to the basic DC attack.

In this paper, we propose a new addition to the DC toolkit, which we call *meet-in-the-filter* (MiF). This technique is especially suitable for ciphers with a slow or incomplete diffusion layer such as the ones based on Addition-Rotation-XOR (ARX). The main idea of the MiF technique is to stop the difference propagation earlier in the cipher resulting in a distinguisher on a fewer number of rounds (smaller value of r) with a relatively high probability p. This comes at the expense of a deeper analysis phase in the bottom rounds, i.e., a relatively high value of u. More specifically, in the MiF technique, we split $u = s + t$ into a precomputed *cluster* of differences for s rounds, then perform a Matsui-like search from the ciphertext difference, running backwards for t rounds up to the meeting point with the difference cluster. The filter discards a pair as wrong if the meeting point (the meet-in-the-filter) does not produce a valid $(s + t)$-round trail. For the reverse search, we use the fact that a differential $(\alpha, \beta \to \gamma)$ has the same probability through modular addition and modular subtraction (see Lemma 3). As a result, MiF produces a set of trails that can be used with an auxiliary key-recovery procedure.

To illustrate the practical use of the MiF technique we apply it to the ARX-based family of block ciphers SPECK. After obtaining the set of 4-round trails produced by MiF, an attacker can use a key recovery procedure similar to the one described by Dinur in [Din14; Din14][1] by just applying it twice – once for the bottom two rounds and once for the penultimate two rounds. Dinur suggested that although counting techniques could be applied to his procedure, it was not likely to improve the complexity of the attack. However, since MiF proposes trail differences for the full four rounds, we can use an advanced key recovery method with *dynamic counting* to improve time complexity.

Given a set of 4-round trails for SPECK, the dynamic counting procedure returns a set of candidate subkeys that satisfies at least c trails. Enforcing this requirement amplifies the filtering of subkey candidates, which reduces the key recovery time. Further, we describe the recursive implementation of the procedure which reduces the memory overhead of counting. This technique is applied to recover the four bottom subkeys of SPECK, which are sufficient to recover the full master key by applying SPECK's key schedule in reverse. An important distinction of our approach to Dinur's [Din14] is that the latter analyzes the bottom four rounds of SPECK$2n$ by making 2^{2n} key guesses for the bottom two of the four rounds (since the difference propagation in these rounds is not known) while in our case, the key-recovery procedure runs on all the four rounds.

We give a detailed analysis of MiF on 11 rounds of SPECK32, using various $1+r+s+t$ round splits. Notably, we have an 11-round MiF attack with practical analysis and data complexities of $2^{24.66}$ and $2^{26.70}$ respectively. Our attack is estimated to be 2^{13} times faster than the deep learning approach introduced in CRYPTO 2019 [Goh19] which has the best-known attack on 11-round SPECK32 with time complexity of 2^{38} for a 50% success rate. Experimentally, our attack recovers the right key in under a second on a PC with a success probability of $\approx 63\%$. We provide experimental verification of the estimated complexities for 11- and 12-round attacks[2]. We also further improve the time complexities of the best attacks reported in the literature on up to 15 rounds of SPECK32 and up to 20 rounds of SPECK64/128.

The outline of the paper is as follows. Section 5.2 reviews previous attacks on SPECK,

[1] We refer to [Din14], which is the extended version of [Din14] and which contains a full description of Dinur's algorithm.

[2] Experimental verification of our 11- and 12-round attacks on SPECK32/64 is available at github.com/1d50f/MiF. Our attack experiments were run on a single core of a laptop with Intel® Core™ i7-1185G7 CPU clocked at 3.00GHz and 32 GiB RAM.

Notation	Definition
FE	Full (Speck) encryptions (time complexity measure)
n	Word size in bits
Speck-$2n/(kn)$	Speck with block size $2n$ and key size kn
$\oplus$, $\wedge$, $\vee$, $\neg$	Bitwise XOR, AND, OR, and NOT
$a \lll b$, $a \ggg b$	Cyclic shift of a by b bits to the left and to the right respectively
ADD , $+$	Addition modulo 2^n
SUB , $-$	Subtraction modulo 2^n
$\|S\|$	Size of the set S
a_i	i-th bit in the big-endian word a, where a_0 is the LSB
R	Total number of rounds
l, r, u	Number of top, middle, bottom rounds in an $l + r + u$ attack
$u = s + t$	Split of bottom rounds into s and t rounds
k	Number of key recovery rounds/key words
α, β, γ	XOR differences for addition or subtraction modulo 2^n
ΔX	XOR difference
$\Delta_{\text{IN}}, \Delta_{\text{OUT}}$	Input and output XOR difference to a differential (trail)
$\tau_r = (\Delta_{\text{IN}} \xrightarrow{r} \Delta_{\text{OUT}})$	A differential (trail) on r rounds
$\mathbf{xdp}^+$, $\mathbf{xdp}^-$	XOR differential probability of addition and subtraction
w, weight	Negative $\log_2$ of differential probability, i.e. $\Pr = 2^{-w}$
$\mathcal{S}(s, w_s)$ or $\mathcal{S}$	Cluster of trails on s rounds with $\Pr \geq 2^{-w_s}$
$\mathcal{T}(t, w_t)$ or $\mathcal{T}$	Set of filtered trails on t rounds with $\Pr \geq 2^{-w_t}$
p, q	Trail (single/differential/cumulative) probabilities
D	Number of chosen plaintexts
n_{trails}	Number of trails returned by MiF
d	Current depth visited by the dynamic key recovery procedure
$B(k; n, p)$	The binomial distribution, $B(k; n, p) = \binom{n}{k} p^k (1-p)^{n-k}$

Table 5.1: Common notations used in this chapter

while Section 5.3 provides basic definitions, theorems, and lemmas used in the paper, as well as some relevant known results. It also includes a high-level description of the Speck family of block ciphers. Section 5.4 presents the Meet-in-the-Filter (MiF) technique followed by the improved key-recovery framework based on counting. Attacks on Speck32 and Speck64/128 using the MiF tool are presented in Section 5.5, Section 5.6 and Section 5.7. Finally, Section 5.8 concludes the paper. The notation used throughout this paper is given in Table 5.1.

5.2 Previous Work

All previous differential attacks on Speck start from a differential (trail) on r rounds to which 1 round is added at the top and u rounds are added at the bottom. In all cases, we can add this additional round at the top due to the fact that the key addition with the first round key is executed at the end of the first round, and so does not influence the attack complexity. Previous attacks on Speck32 and Speck64/128 along with the proposed new attacks are listed in Table 5.2. Time complexity is measured in the number of full encryptions (FE), data complexity in the number of chosen plaintexts, and memory is in bytes. A brief summary follows next, which covers classical differential attacks and recently proposed differential-neural approaches.

Variant	Rounds	Split	Pr diff	Time	Data	Mem	Ref
SPECK32/64	11/22	1+6+4	2^{-13}	2^{46}	2^{14}	2^{22}	[Din14]
SPECK32/64	11/22	1+9+1	Neural	2^{38}	$2^{14.5}$	-	[Goh19]
SPECK32/64	12/22	1+7+4	2^{-18}	2^{51}	2^{19}	2^{22}	[Din14]
SPECK32/64	12/22	1+9+1	Neural	$2^{43.40}$	$2^{22.97}$	-	[Goh19]
SPECK32/64	12/22	1+10+1	Neural	$2^{44.89}$	2^{22}	-	[Bao+21]
SPECK32/64	13/22	1+8+4	2^{-24}	2^{57}	2^{25}	2^{22}	[Din14]
SPECK32/64	14/22	1+9+4	2^{-30}	2^{63}	2^{31}	2^{22}	[Din14]
SPECK32/64	14/22	1+9+4	$2^{-29.47}$	$2^{62.47}$	$2^{30.47}$	2^{22}	[SHY16]
SPECK32/64	15/22	1+10+4	$2^{-30.39}$	$2^{63.39}$	$2^{31.39}$	2^{22}	[Lee+18]
SPECK32/64	11/22	1+0+8+2	-	$2^{40.15}$	$2^{14.11}$	$2^{28.97}$	this paper
SPECK32/64	11/22	1+0+8+2	-	$2^{34.87}$	$2^{15.58}$	$2^{24.71}$	this paper
SPECK32/64	11/22	1+0+8+2	-	$2^{24.66}$	$2^{26.70}$	$2^{22.02}$	this paper
SPECK32/64	12/22	1+0+9+2	-	$2^{45.91}$	$2^{18.88}$	$2^{32.13}$	this paper
SPECK32/64	12/22	1+7+2+2	$2^{-29.85}$	$2^{41.97}$	$2^{22.45}$	$2^{30.46}$	this paper
SPECK32/64	12/22	1+8+1+2	2^{-24}	$2^{33.84}$	$2^{30.42}$	$2^{24.75}$	this paper
SPECK32/64	13/22	1+0+10+2	-	$2^{56.41}$	$2^{25.27}$	$2^{36.85}$	this paper
SPECK32/64	13/22	1+8+2+2	$2^{-23.85}$	$2^{50.16}$	$2^{31.13}$	$2^{31.07}$	this paper
SPECK32/64	14/22	1+9+2+2	$2^{-29.37}$	$2^{61.35}$	$2^{30.64}$	2^{22}	this paper
SPECK32/64	14/22	1+9+2+2	$2^{-29.37}$	$2^{60.99}$	$2^{31.75}$	$2^{41.91}$	this paper
SPECK32/64	15/22	1+10+2+2	$2^{-30.39}$	$2^{62.25}$	$2^{31.39}$	2^{22}	this paper
SPECK64/128	13/27	1+8+4	2^{-29}	2^{96}	2^{30}	2^{22}	[Din14]
SPECK64/128	15/27	1+13+1	$2^{-58.9}$	$2^{61.1}$	2^{61}	2^{32}	[Abe+14]
SPECK64/128	16/27	1+14+1	$2^{-59.02}$	2^{80}	2^{63}	-	[BRV14]
SPECK64/128	19/27	1+14+4	2^{-60}	2^{125}	2^{61}	2^{22}	[Din14]
SPECK64/128	20/27	1+15+4	$2^{-60.56}$	$2^{125.56}$	$2^{61.56}$	2^{22}	[SHY16]
SPECK64/128	13/27	1+8+2+2	$2^{-28.87}$	$2^{59.53}$	$2^{31.46}$	$2^{39.97}$	this paper
SPECK64/128	13/27	1+8+2+2	$2^{-28.87}$	$2^{52.45}$	$2^{32.14}$	$2^{39.70}$	this paper
SPECK64/128	19/27	1+14+2+2	$2^{-55.69}$	$2^{101.08}$	$2^{61.03}$	$2^{67.30}$	this paper
SPECK64/128	20/27	1+15+2+2	$2^{-60.73}$	$2^{122.69}$	$2^{63.96}$	$2^{77.19}$	this paper

TABLE 5.2: Summary of differential attacks on SPECK 32 and SPECK64/128. **Rounds** R/R' denotes that R out of R' rounds are attacked; **Split** $l + r + k = R$ denotes that to an initial differential (trail) on r rounds, l rounds are added at the top and k rounds are added at the bottom; **Pr diff** is the probability of the differential (trail) on r rounds. **Time**, **Data**, **Mem** are resp. the time, data and memory complexity of the attack; **Ref** is the reference to the publication describing the attack. Highlighted cells indicate the best attack time complexities.

In SAC 2014, Dinur proposed new attacks on SPECK32 for up to 14 rounds, with the latter having time T and data D complexities of $(T, D)_{14R} = (2^{63}, 2^{31})$ [Din14]. Later, in CRYPTO 2019, Gohr showed that neural networks could be trained to be cryptographic distinguishers [Goh19]. His 11-round attack on SPECK32 uses differential-neural distinguishers that consist of 7-round (and a 6-round) neural distinguisher appended to a 2-round classical differential. The attack has a success rate of about 50% to recover the final 2 subkeys[3] with $(T, D)_{11R} = (2^{38}, 2^{14.5})$. Using a similar

[3]The second subkey was allowed to be wrong for at most 2 bits.

attack procedure, Gohr also has a 12-round attack with $(T, D)_{12R} = (2^{43.40}, 2^{22.97})$ but only a 40% success rate. Benamira *et al.* later delved into the inner workings of Gohr's approach from the standpoint of classical differential cryptanalysis [Ben+21]. They found that these distinguishers rely not only on the ciphertext pair but also on the difference distributions in the bottom two rounds. Apart from being able to better interpret the behaviour of the neural distinguishers and improving their accuracy, no new attacks on Speck32 were reported. In [Bao+21], Bao *et al.* use a 10-round differential-neural distinguisher to mount a 12-round key recovery attack on Speck32 using a similar key recovery framework as Gohr. By using more than one differential prepended to the neural distinguisher, they reported an attack with $(T, D)_{12R} = (2^{44.89}, 2^{22})$ and a higher success rate of 86%. Going back to classical differential cryptanalysis, Song *et al.* [SHY16] and Lee *et al.* [Lee+18] reported attacks on 14 and 15 rounds of Speck32 with resp. $(T, D)_{14R} = (2^{62.47}, 2^{30.47})$ and $(T, D)_{15R} = (2^{63.39}, 2^{31.39})$ by using differentials rather than single trails as their distinguishers.

Next, we take a look at past attacks on Speck64/128. In [Abe+14], Abed *et al.* use a differential trail on 13 rounds to which they add one round at the top and at the bottom to mount a $1 + 13 + 1$ attack on Speck64/128. During the same period, Biryukov *et al.* [BRV14] reported an attack with time and data complexities $(T, D)_{16R} = (2^{80}, 2^{63})$ for Speck64/128. In [Din14], Dinur mounts a $1+14+4$ attack on Speck64/128 with $(T, D)_{19R} = (2^{125}, 2^{61})$. Song *et al.* [SHY16] attack 20-round Speck64/128 with $(T, D)_{20R} = (2^{125.56}, 2^{61.56})$. This was a $1 + 15 + 4$ attack that used a differential (rather than a single trail) for 15 rounds with $\Pr = 2^{-60.56}$ (the single trail probability is 2^{62}). The latter results in a slight improvement, the rest being the same as in Dinur's attack. Complexity-wise Song *et al.* attacks are already close to biclique attacks which work almost for any cipher.

The proposed MiF technique bears some similarity to earlier results e.g. on DES [DSP07], AES [DKS10; DFJ13] and LowMC [RST18; LIM21]. In [DSP07], the authors similarly lower the data complexity of their attack by recovering internal values rather than key bits (in contrast, we recover internal differences). In [DKS10], by enumerating the possible differential input/outputs to active S-boxes, a set of possible differential trails is recovered. The same idea is built upon in some of the results in [DFJ13]. More recently, attacks on LowMC [RST18; LIM21] leverage upon a conceptually similar reconstruction of differential trails but only for probability-one trails.

Techniques for the automatic search for differential trails for Speck can be broadly divided into two groups. In the first group the problem is represented in terms of Mixed Integer Linear Programming (MILP) or Satisfiability Modulo Theory (SMT) and off-the-shelf MILP or SAT solvers are employed to execute the search. Some results in this group are by Fu *et al.* [Fu+16] (MILP) and Song *et al.* [SHY16] (SMT) with the latter applying the method proposed by Mouha *et al.* [MP13] to construct a long differential trail from two short ones. The second group is composed of dedicated techniques based on Matsui's search algorithm [Mat94]. Biryukov *et al.* [BVC16] proposed the first adaptation of this algorithm to ARX ciphers and an optimised version using carry-bit-dependent difference distribution tables (CDDT) was later developed by Liu *et al.* [Liu+19]. Huang *et al.* [HW19] further optimized the latter using combinatorial DDT (cDDT). We note that the differential search algorithms from [BVC16; HW19; Liu+19] are complete, i.e., given enough time, they will return all the differential trails with a given differential probability.

5.3 Preliminaries

We begin with some preliminaries, necessary to understand the main results presented in subsequent sections. In the following exposition, addition and subtraction modulo 2^n are denoted respectively by ADD and SUB.

5.3.1 Differential Cryptanalysis

Differential cryptanalysis analyzes pairs of encryptions $P_1 \mapsto C_1, P_2 \mapsto C_2$ by studying the propagation of the input difference $\Delta P = P_1 \oplus P_2$ to the output difference $\Delta C = C_1 \oplus C_2$ through the cipher, which is known as a differential characteristic or trail. A differential trail consists of a sequence of differences:

$$\Delta P \to \delta_1 \to \delta_2 \to \ldots \to \delta_{\mathsf{r}-1} \to \Delta C. \tag{5.1}$$

To perform an attack, an adversary needs a differential trail with sufficiently high differential probability:

$$p = \Pr_P[\Delta P \to \ldots \to \Delta C], \tag{5.2}$$

which is defined as the probability over all plaintexts. However, for simplicity of the analysis and due to the presence of round keys in ciphers, it is usually approximated by the probability of the trail over assumed-to-be-independent round keys (the so-called Markov assumption [LM91]). In that case, the probability of the trail can be computed simply as the product of the probabilities of all the individual transitions:

$$p = \Pr[\Delta P \to \delta_1] \cdot \Pr[\delta_1 \to \delta_2] \cdot \ldots \cdot \Pr[\delta_{\mathsf{r}-1} \to \Delta C]. \tag{5.3}$$

A better estimate of the differential probability can be obtained by collecting all differential trails that have the same input and output differences:

$$p = \Pr[\Delta P \to \Delta C] = \sum_{\delta_1 \ldots \delta_{\mathsf{r}\text{-}1}} \Pr[\Delta P \to \delta_1 \to \ldots \to \delta_{r-1} \to \Delta C]. \tag{5.4}$$

The *weight* of a differential (trail) is defined as $w = -\log_2(p)$. The following variant of the Markov assumption is used to analyze our attack time complexities.

Assumption 1. *For a (possibly truncated) differential trail $\Delta P \to \Delta C$ with a weight w, and a uniformly and independently sampled pair of ciphertexts (C_1, C_2), the average* fraction *of subkeys for which the partial decryption of (C_1, C_2) follows the trail is equal to 2^{-w}.*

5.3.2 The Differential Probability of ADD and SUB

The differential probabilities of addition/subtraction modulo 2^n were studied by Lipmaa and Moriai [LM01].

Definition 3. $\mathbf{xdp}^+$ and $\mathbf{xdp}^-$ are the probabilities with which input XOR differences α, β propagate to output XOR difference γ through the operations ADD and SUB respectively, computed over all n-bit inputs a, b:

$$\mathbf{xdp}^+(\alpha,\beta,\gamma) = \quad 2^{-2n} \cdot |\{(a,b) : ((a \oplus \alpha) + (b \oplus \beta)) \oplus (a + b) = \gamma\}| \,, \tag{5.5}$$

$$\mathbf{xdp}^-(\alpha,\beta,\gamma) = \quad 2^{-2n} \cdot |\{(a,b) : ((a \oplus \alpha) - (b \oplus \beta)) \oplus (a - b) = \gamma\}| \,. \tag{5.6}$$

Lemma 1 ([LM01, Lemma 3])**.** *The probability* $\mathbf{xdp}^{+}(\alpha,\beta,\gamma)$ *is non-zero if and only if*

$$\alpha_i \oplus \beta_i \oplus \gamma_i = \begin{cases} 0 & \text{if} \quad (i = 0) \ , \\ \beta_{i-1} & \text{if} \quad (i \geq 1) \wedge (\alpha_{i-1} = \beta_{i-1} = \gamma_{i-1}) \ . \end{cases} \tag{5.7}$$

If the probability $\mathbf{xdp}^{+}(\alpha,\beta,\gamma)$ is non-zero (i.e., the differential $(\alpha,\beta \to \gamma)$ is possible), its exact value can be computed with the formula given in Theorem 2.

Theorem 1 ([LM01, Algorithm 2])**.** *If* $\mathbf{xdp}^{+}(\alpha,\beta,\gamma) > 0$ *then*

$$\mathbf{xdp}^{+}(\alpha,\beta,\gamma) = 2^{-n+l+1} : \ l = |\{i \leq n-1 : \ \alpha_i = \beta_i = \gamma_i\}| \ . \tag{5.8}$$

Note that the maximum possible transition weight through ADD is $n-1$. From Lemma 9 and Theorem 2, we can deduce that the differential probability of transitions through ADD is equal to those through SUB, and does not depend on the order of the three differences.

Lemma 2. *The probability* $\mathbf{xdp}^{+}(\alpha,\beta,\gamma)$ *is invariant under any permutation of the inputs* α,β,γ*, i.e.,*

$$\mathbf{xdp}^{+}(\alpha,\beta,\gamma) = \mathbf{xdp}^{+}(\alpha,\gamma,\beta) = \mathbf{xdp}^{+}(\beta,\alpha,\gamma) = \dots \ . \tag{5.9}$$

Lemma 3. *The differential* $(\alpha,\beta \to \gamma)$ *has the same probability through modular addition and modular subtraction for any choice of differences* α,β,γ*, i.e.,*

$$\mathbf{xdp}^{+}(\alpha,\beta,\gamma) = \mathbf{xdp}^{-}(\alpha,\beta,\gamma) \ . \tag{5.10}$$

5.3.3 Distribution of Differential Weights and Probabilities of ADD

In this section, we recall and derive properties of the distribution of weights and/or probabilities of differential transitions through the ADD operation. These properties will be used in the analysis of the MiF tool and complexities of the attacks. All proofs can be easily derived from the following lemma by Lipmaa and Moriai [LM01] and are omitted due to page limitations.

Lemma 4 ([LM01, Theorem 2])**.** *The fraction of all transitions through ADD (including invalid ones) having weight* w *is given by*

$$\Pr_{\alpha,\beta,\gamma}[\mathbf{xdp}^{+}(\alpha,\beta,\gamma) = 2^{-w}] = \frac{1}{2}\left(\frac{7}{8}\right)^{n-1} B(w; n-1, \frac{6}{7}) \ . \tag{5.11}$$

Lemma 5. *Let* α,β *be chosen independently and uniformly at random. The expected number of differences* γ *such that the differential transition* $(\alpha,\beta) \to \gamma$ *is valid (i.e.,* $\mathbf{xdp}^{+}(\alpha,\beta,\gamma) > 0$*) is given by*

$$\mathop{\mathrm{E}}_{\alpha,\beta}[|\{\gamma : \mathbf{xdp}^{+}(\alpha,\beta,\gamma) > 0\}|] = \left(\frac{7}{4}\right)^{n-1} = 2^{(n-1)\log_2(\frac{7}{4})} \ . \tag{5.12}$$

Lemma 6. *Let* $(\alpha,\beta) \to \gamma$ *be a transition through ADD sampled uniformly at random from all valid transitions through ADD. The expected differential transition* weight w

is equal to

$$\mathop{\mathrm{E}}_{\substack{\alpha,\beta,\gamma:\\ \mathbf{xdp}^+(\alpha,\beta,\gamma)>0}}[-\log_2(\mathbf{xdp}^+(\alpha,\beta,\gamma))] = \frac{6}{7}(n-1)\ . \tag{5.13}$$

Lemma 7. *Let $(\alpha,\beta)\to\gamma$ be a transition through ADD sampled uniformly at random from all valid transitions through ADD. The average differential transition* probability *p is given by*

$$\mathop{\mathrm{E}}_{\substack{\alpha,\beta,\gamma:\\ \mathbf{xdp}^+(\alpha,\beta,\gamma)>0}}[\mathbf{xdp}^+(\alpha,\beta,\gamma)] = \left(\frac{4}{7}\right)^{n-1}\ . \tag{5.14}$$

Example 1. SPECK32 uses 16-bit additions, for which the differential transitions have average weight approximately $w = 12.86$ and average probability approximately $2^{-12.11}$. SPECK64 uses 32-bit additions, for which the differential transitions have average weight approximately $w = 26.57$ and average probability approximately $2^{-25.03}$.

5.3.4 Dinur's Attack

Since our work draws parallels to Dinur's attack, we describe it briefly in this section. In its basic version, Dinur's attack uses an r round differential to attack $r+2$ rounds. All internal differences and some values in the bottom two rounds are known from the differential and ciphertexts. To recover the remaining unknown internal values, Dinur applies a guess-and-determine strategy that works bitwise on the bottom two modular additions (cf. *1RProcedure, 2RProcedure* [Din14, Appendix A]). As a result, the last two round keys are recovered. The basic $r+2$ attack is then trivially extended to $r+4$ rounds for any SPECK variant by recovering two additional round keys through an exhaustive search, which increases the attack complexity by a factor of 2^{2n}. Dinur's attack applies two filtration procedures, called *one-bit* and *multi-bit* filters [Din14, § 7.2], that exploit the differential properties of modular addition. The one-bit filter provides the main filtration power and is effectively a check for the conditions of Lemma 9. It gives filtration efficiency of $\frac{1}{2}\cdot(\frac{7}{8})^{n-1}\approx 2^{-7}$ for each 32-bit ADD with known α,β,γ differences. The multi-bit filter provides further improvement by a factor of about 2^{-3} for each 32-bit ADD operation.

5.3.5 Description of the Block Cipher Speck

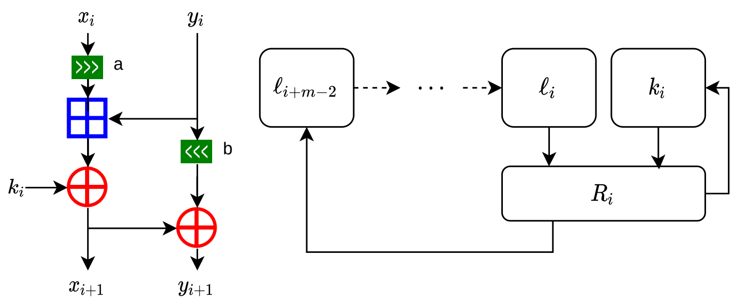

FIGURE 5.1: Speck round function and key schedule.

SPECK is a family of lightweight block ciphers proposed by USA National Security Agency in [Bea+13]. It follows an iterative ARX design, supporting block sizes of $32, 48, 64, 96$, and 128 bits and various key sizes. The members of the SPECK family have been designed to provide good performance in software with their main target being microcontrollers. With SPECK$2n/(kn)$ is denoted the instance of SPECK with a block size of $2n$ bits composed of two n-bit words and a key size of kn bits, where k denotes the number of keywords. SPECK$2n$ uses three operations over n-bit words: bitwise XOR, addition modulo 2^n and bitwise rotation. The key-dependant round function of SPECK$2n$ depicted in Figure 5.1 is a map $R_K : \{0,1\}^{2n} \to \{0,1\}^{2n}$ defined as

$$R_K(x,y) = ((x \ggg r_a) + y) \oplus K, (y \lll r_b) \oplus (((x \ggg r_a) + y) \oplus K) \ , \tag{5.15}$$

where the rotation values are $r_a = 7, r_b = 2$ for $n = 16$, and $r_a = 8, r_b = 3$ for all other block sizes. The decryption of SPECK uses modular subtraction on the inverted round function and is naturally derived. The key schedule of SPECK$2n$ takes the master key and generates R round-key words $K_0, K_1, \cdots, K_{R-1}$, where R is the number of rounds, using the same round function as used by the encryption. For a detailed description of SPECK we refer the reader to [Bea+13].

5.4 The Meet-in-the-Filter (MiF) Attack

In this section, we describe the Meet-in-the-Filter (MiF) attack, which is divided into two main parts – the MiF tool and the key recovery procedure based on dynamic counting. It is applicable to ciphers with incomplete or relatively slow diffusion such as ARX.

5.4.1 The MiF Tool

Consider a block cipher with $r+u$ rounds split into r rounds covered by a differential (trail) and u rounds covered by backward search. The goal of the MiF *tool* is to efficiently enumerate trails for the bottom u rounds. We can further split u into two parts: $u = s+t$, in order to obtain a time-memory trade-off. The s and t rounds of the split are processed separately in search of a meeting point (a matching difference). An illustration of the MiF filter is shown in Figure 5.2. We start from an r-round

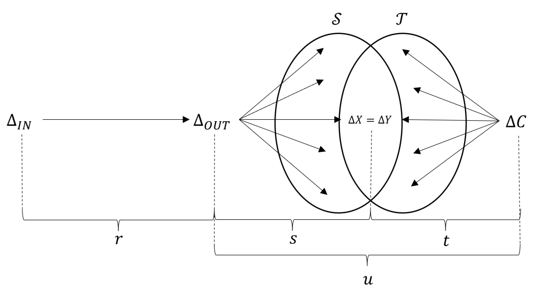

FIGURE 5.2: Illustration of MiF with an $r+u$ and $u = s+t$ split.

differential with probability p denoted as $\Delta_{\text{IN}} \xrightarrow{r} \Delta_{\text{OUT}}$. Next we choose a suitable split of u into s top and t bottom rounds ($u = s+t$) together with corresponding probability thresholds 2^{-w_s} and 2^{-w_t}. In an offline phase, we apply Huang *et al.*'s Matsui-like search [HW19] to prepare *the cluster*.

Definition 4. The *cluster* $\mathcal{S}(s, w_s)$ is the set of all s-round trails τ_s starting with the difference Δ_{OUT} and having probability at least 2^{-w_s}:

$$\mathcal{S}(s, w_s) = \left\{ \tau_s = (\Delta_{\text{OUT}} \xrightarrow{s} \Delta X) : \ \Pr[\tau_s] \geq 2^{-w_s} \right\} \ . \tag{5.16}$$

The functional notation $\mathcal{S}(s, w_s)$ stresses the fact that the set $\mathcal{S}$ is a function of the parameters s and w_s. We use $\mathcal{S}$ as a shorthand for $\mathcal{S}(s, w_s)$ when the parameters are clear from the context. Constructing $\mathcal{S}$ would usually require negligible precomputation time compared to the full differential attacks.

In the online phase, a set of $c' \cdot p^{-1}$ (for some small constant $c' \geq 1$) chosen plaintext pairs $(P_1, P_2 = P_1 \oplus \Delta_{\mathrm{IN}})$ are encrypted for $r + u$ rounds. For each corresponding ciphertext pair (C_1, C_2), a reverse search on t rounds starting with the ciphertext difference $\Delta C = C_1 \oplus C_2$ as input is executed. Since the reverse search is performed on Speck in decryption mode, the modular addition operation ADD is replaced by modular subtraction SUB which does not change the probability computation due to Lemma 3. For a given observed ciphertext pair, the reverse search produces the filter-set $\mathcal{T}(t, w_t)$.

Definition 5. The *filter-set* $\mathcal{T}$ consists of all t-round trails τ_t starting from ΔC in the reverse direction and having probability at least 2^{w_t}.

$$\mathcal{T}(t, w_t) = \left\{ \tau_t = (\Delta C \xrightarrow{t} \Delta Y) : \ \Pr[\tau_t] \geq 2^{-w_t} \right\} \ . \tag{5.17}$$

Similarly to $\mathcal{S}(s, w_s)$, the set $\mathcal{T}(t, w_t)$ is expressed as a function of the parameters t and w_t. We use $\mathcal{T}$ as a shorthand for $\mathcal{T}(t, w_t)$ when the parameters are clear from the context. Of all trails τ_t in $\mathcal{T}$, we keep only the ones whose output difference ΔY matches an output difference ΔX of a trail τ_s in $\mathcal{S}$. A match between a given $\tau_s \in \mathcal{S}$ and a given $\tau_t \in \mathcal{T}$ results in a u-round trail τ_u obtained by the following concatenation:

$$\tau_u = (\tau_t || \tau_s) = \Delta C \xrightarrow{u} \Delta_{\mathrm{OUT}} = \left(\Delta \mathrm{C} \xrightarrow{t} (\Delta Y = \Delta X) \xrightarrow{s} \Delta_{\mathrm{OUT}} \right) . \tag{5.18}$$

A ciphertext pair for which a match is found, is recorded as a candidate *right pair*, i.e., a pair whose corresponding plaintexts (P_1, P_2) have followed the differential (trail) $(\Delta_{\mathrm{IN}} \xrightarrow{r} \Delta_{\mathrm{OUT}})$. Each such pair comes with a set of suggested u-round trails $\{\tau_u = \Delta_{\mathrm{OUT}} \xrightarrow{s+t} \Delta C\}$. The latter contains information for the key-recovery phase and is passed on to the key-recovery procedure. The set $\mathcal{S}$ is referred to as *the cluster* while the process of matching the set $\mathcal{T}$ against $\mathcal{S}$ is referred to as *the (backward) filter*. The absolute values of the logarithm base-2 probability thresholds – i.e., the constants w_s and w_t – are called respectively the *cluster weight* and the *filter weight*. Since the split $s + t$ can be seen as one large u-round filter that passes only candidate right pairs, the procedure is called *meet-in-the-filter* or MiF. In general, MiF offers the attacker a reduction in filtration complexity for the u bottom rounds through a time-memory trade-off.

5.4.1.1 Efficiency and Loss Factor

Pairs of plaintexts (P_1, P_2) that follow the differential for the top r rounds are called *right pairs* or *signal* while those that do not are *wrong pairs* or *noise*. After the application of MiF, some of this signal may be lost due to the weight thresholds (w_s and w_t) being applied to the bottom u rounds. Denote the probability that a right pair follows a u-round trail produced (or filtered) by MiF by q. Such a u-round trail is called a *right trail*, i.e., a right u-round trail is one that will be followed by the corresponding right pair after going through the initial r-round differential. We refer to q as the *efficiency* of the MiF filter, the inverse of which, q^{-1}, is called the *loss* *factor*. The latter is the value by which the attacker needs to multiply the initial data $D = 2 \cdot c' \cdot p^{-1}$ to compensate for the decreased filter efficiency. The constant c' maintains the probability of catching at least c right trails in the set of trails (dataset) produced by MiF, as required by our key recovery technique. Thus the overall data complexity of the attack is a function of the efficiency of the MiF filter and is equal to Dq^{-1}.

The efficiency of the MiF filter depends on the choice of the split values s and t, and the corresponding cluster and filter weights, w_s and w_t respectively. To maximize efficiency, the filter weight must be set large enough to allow all possible difference propagations in the backward filter. Based on Lemma 6, the average weight of a random valid t-round trail can be estimated as $\frac{6}{7}(n-1)t$, e.g. for $t = 2$ and $n = 16$, the average weight of a 2-round trail is 25.71. To ensure that no trails will be discarded by the backward filter, the maximum value $w_t = 2(n-1) = 30$ should be set. If no limit is imposed on the backward filter, we can estimate q as the cumulative probability of all trails in the $\mathcal{S}$ that comes from one r-round differential:

$$q = \sum_{\tau_s \in \mathcal{S}} \Pr[\tau_s] \ . \tag{5.19}$$

Typically, most trails suggested by MiF will not be right (i.e., are noise). We will denote the number of trails returned by MiF as n_{trails}. These trails, together with respective ciphertext pairs, are passed on to the key recovery stage, which we will describe and analyze in the following section.

5.4.2 Key Recovery using Single-Trail Analysis

In this section, we describe a general key recovery procedure based on *single* trail analysis. We recall the general setting – an attacker uses a differential $\Delta_{\text{IN}} \xrightarrow{r} \Delta_{\text{OUT}}$ over r rounds and queries encryption of a plaintext pair with the difference Δ_{IN} over $r+u$ rounds, obtaining a ciphertext pair (C_1, C_2) with a difference ΔC. MiF suggests a set of valid trails of the form $\Delta C \xrightarrow{u} \Delta_{\text{OUT}}$, with a hypothesis that this set contains the right trail.

In single-trail analysis, the attacker analyzes each proposed trail independently of other encryptions and all other trails. The analysis returns a set of *candidate subkeys* for analyzed $k \leq u$ rounds, for which the partial decryption of the ciphertext pair (C_1, C_2) follows first k rounds of the suggested trail $\Delta C \xrightarrow{u} \Delta_{\text{OUT}}$ (i.e., the subtrail $\Delta C \xrightarrow{k} \Delta Z$ of the trail $\Delta C \xrightarrow{k} \Delta Z \xrightarrow{u-k} \Delta_{\text{OUT}}$). These candidate subkeys can then be used to derive candidates for the master key, to be tested against known encryptions or to follow the expected differential trail. The full key recovery attack simply consists of applying a sufficient number of iterations of the above procedure.

This setting follows the direction of Dinur's work; in fact, the procedure described in this section is simply a generalization of the analysis stage of Dinur's attacks. One of the main advantages of MiF is that this procedure can be applied right from the beginning due to the knowledge of a set of candidate trails. In addition, we pay closer attention to the theoretical analysis of the attack's complexity.

5.4.2.1 Recursive Single-Trail Procedure

The procedure takes as input a ciphertext pair (C_1, C_2) and a trail $\Delta C \xrightarrow{k} \Delta Z$; it outputs all k-round subkeys for which the partial decryption of the pair (C_1, C_2) follows the given trail. The idea is simply to guess the subkeys *in chunks* and *recursively*.

Guessing subkeys *in (small) chunks* allows to quickly filter out wrong guesses, which are those making the partial decryption of the ciphertext pair (C_1, C_2) diverge from the given differential trail. A simple example is guessing subkeys round-by-round: after guessing one full subkey, we may decrypt the pair by one round and check whether the obtained difference follows the trail. Since many ciphers have incomplete diffusion over a small number of rounds, guessing even a small part of the subkey often allows partial decryption and computation of a part of the difference in the previous round, leading to faster discarding of invalid subkeys. For example, guessing even a single subkey bit in Speck (starting from least significant bits) yields one bit of the difference in the previous round. Smaller chunks allow reducing the unnecessary work, bringing the procedure cost close to the theoretical lower bound arising from the output size of the procedure – the total number of valid subkey candidates.

Recursive implementation of the procedure aims at minimizing the memory complexity. Indeed, the total number of candidate subkeys can be huge, and keeping all of them in memory at the same time is unnecessarily costly. Recursive guessing of the subkey chunks allows reducing the memory footprint of the procedure to negligible. An alternative formulation of this method is the depth-first traversal of the search tree (as opposed to breadth-first traversal).

Example 2. In our attacks on Speck variants, we will set the chunk size to be 1 bit. The recursive procedure thus will simply recover the subkeys round-by-round and bit-by-bit, checking the conformance to the differential trail after each subkey bit guess. In Speck, the n-bit subkey κ is XORed right after the ADD operation. When decrypting, the ADD becomes SUB and this subkey hides one of the inputs. Guessing i least significant bits of the subkey κ allows computing SUB on i least significant bits, leading to the knowledge of i least significant bits of the difference in the previous round[4], which can be used as a filter discarding wrong subkey guesses.

The following definition formalizes the notion of truncated trails, i.e., parts of the analyzed trail that can be tested after guessing some subkey chunks.

Definition 6. Given a differential trail τ over k rounds and an integer d, by the **differential trail τ truncated at the depth d** we will understand τ restricted to all bit positions where the difference can be computed from the ciphertext difference and first d chunks of subkeys guessed. The maximum depth $d_{\max}$ is defined as the full number of chunks of subkeys that have to be guessed in the attack.

Example 3. In Speck, the maximum depth $d_{\max}$ is simply equal to the number of key recovery rounds (2, 3 or 4) times the word size, i.e.,

$$d_{\max} = k \cdot n. \tag{5.20}$$

5.4.3 Distributions of Weights in MiF Trails

The complexity of the MiF attack depends significantly on the chosen differential (especially on its output difference Δ_{OUT}), the round split, the cluster and filter weights w_s and w_t. These parameters affect in particular the properties of the trails suggested by MiF, namely the distribution of weights of *truncated trails* (in the sense of Def. 6), which directly affects the time complexity of the attack. Estimating this

[4]In fact, guessing $i < n$ bits allows to compute $i + 1$ bits of the difference. We will use this fact in Claim 2 to reduce the complexity.

distribution purely by using theory from Section 5.3.3 is not possible as the evolution of the weights of truncated trails with depth is not uniform (we will show it on examples of our attacks). The time complexity of the multiple-trail key recovery is especially sensitive to intermediate weights, as they will often define the dominating stages of the attack.

Definition 7. Given an integer d, let q_d denote the average probability for the MiF trails truncated at depth d (the trails are sampled uniformly at random from the possible output of MiF).

By distributions of weights/probabilities in MiF trails we will mean the values $(q_0, q_1, \ldots, q_{d_{\max}})$. In addition, the attack's complexity depends directly on the (expected) number of trails to be suggested by MiF. It is thus necessary to be able to compute these quantities in order to compute the time complexity of the attacks. We describe two methods, a heuristic sampling-based method, and a precise trail enumeration-based method.

5.4.3.1 Method 1: Generic Sampling (Heuristic)

The most straightforward way to obtain the distributions of weights of truncated trails is to partially simulate the attack and obtain a collection of trails from MiF, to be used further to compute the necessary distributions. Running a full attack in most cases can be impractical. However, for sampling, the simulation process can be optimized significantly. An important observation is that the high complexity of the attacks stems from the difficulty of catching the *signal* (right pairs/trails), while most of the attack time is actually spent on *noise* (wrong pairs/trails). Since the absence of a few right trails would not change the distributions noticeably, we can restrict sampling to *noise only.* To this end, we propose the following simple procedure:

1. generate a random ciphertext difference ΔC (or, for more genuine results, encrypt a random plaintext pair following the chosen input difference Δ_{IN});
2. run the MiF tool and obtain a set of suggested trails;
3. update the required distributions from the given set;
4. repeat from Step 1 until sufficient precision is reached.

In our attacks on Speck32 and Speck64 we noticed that sampling provides surprisingly stable and precise results. Our usual sampling goal is 1 million trails, or less for very low cluster weights w_s, where a large number of encryptions is needed to pass through the MiF tool. For these low cluster weight/small cluster scenarios, we can instead use Method 2. For larger cluster weights w_s, one has to ensure that a large number of different ciphertext differences is involved, since a collection of 1 million trails suggested from just a couple of encryptions would not be sufficiently representative. In addition, sampling allows estimating well the average number of trails suggested by MiF per one encrypted pair. This is vital for computing the expected number of trails in a concrete attack, which in turn is needed to compute the time complexities (Claim 1, 2 and 3 below).

5.4.3.2 Method 2: Cluster Trail Enumeration (Precise)

While the first method starts from a (random) ciphertext difference ΔC (with the motivation of obtaining representative samples), another option is to start from the computed cluster $\mathcal{S}$. For small clusters and small numbers of filter rounds, we can efficiently enumerate *all* possible trails that can be suggested by MiF. This procedure can be equivalently described as extending the cluster to cover the backward filter rounds as well. Since each possible ciphertext difference ΔC is equally possible (for the overwhelming majority of the wrong pairs), each possible trail from MiF is equally possible to be returned. Therefore, the required distributions can be computed directly from this set of trails as if it was sampled as in Method 1. Here, we use the linearity of expectation which does not require independence. For example, if multiple trails end in the same difference ΔC, they will be always together suggested (or not) by MiF, but this does not affect the *expected* probabilities of truncated trails.

Similarly, the average number of suggested trails per encrypted pair can be computed directly as the number of trails in this "extended cluster" divided by the size $2^{|C|}$ of the ciphertext space. Indeed, each such trail adds one suggested trail when the respective ciphertext difference is hit, and the expected total value of suggested trails can, by the linearity of expectation, be computed as a sum over all trails, which acts simply as multiplication by the number of possible trails.

5.4.4 Key Recovery Complexity Analysis

We begin with the complexity analysis of the single-trail case, and we will build the analysis of the multiple-trail case on top of it. Our estimations will be based on the MiF trail weight distributions computed using techniques described in Section 5.4.3. For simplicity and due to relevance for Speck, we will assume the chunk size of 1 bit. Our key instrument is the following lemma, which connects the distribution q_d of weights/probabilities of truncated trails and the number of surviving trail-subkey pairs per depth.

Lemma 8. *At depth d of the single-trail procedure, across all branches and n_{trails} initial trails, there are on average*

$$v_d = n_{\text{trails}} \cdot 2^d \cdot q_d \tag{5.21}$$

trail-subkey pairs visited.

Proof. Follows as an application of Assumption 1 to the key recovery procedure[5]. □

The total time complexity T_{cnt} of the key recovery procedure splits into two major parts: the complexity T_{enum} of enumerating (recursively) the subkey candidates and the complexity T_{trials} of checking the candidates by partial trial decryptions:

$$T_{\text{cnt}} = T_{\text{enum}} + T_{\text{trials}}. \tag{5.22}$$

5.4.4.1 Estimating T_{trials}

The time complexity of the trial decryptions can be easily derived from the number of the final subkey candidates $v_{d_{\max}}$ suggested by the key recovery procedure. Naturally, we also assume that the key schedule can be easily inverted and all rounds' subkeys can be computed from the recovered subkey candidates (this is the case for most modern ciphers), at the cost proportional to the number of involved rounds, namely, $\frac{R-k}{R}$ FE.

[5]Even though Speck is known not to be a Markov cipher, the theory holds well in practice as confirmed by our experiments.

In cases when the differential trail is known for at least 1 round longer than the key recovery requires, it can be used to test a subkey candidate at the lower cost of 2 round decryptions (equal to $\frac{2}{R}$ FE). Note that one-round trail extension with even a relatively low weight (say, 5) filters out most of the wrong candidates (31/32) and the consequent rounds add negligible complexity. This was suggested already in Dinur's work [Din14, Section 6], but since it did not affect the dominating parts of his attacks, it was left only as a suggestion. However, this shortcut might not be available if we have used differential rather than a single trail.

Claim 1. *Under the above assumptions,*

$$T_{\text{trials}} \leq v_{d_{\max}} \cdot \frac{R'}{R} \quad FE, \tag{5.23}$$

where $R' = 2$ if the differential trail is known for at least one more round, and $R' = R - k$ otherwise.

5.4.4.2 Estimating T_{enum} (Single-Trail)

In order to estimate T_{enum}, we will assume that the time complexity of the single-trail recursive procedure is overwhelmingly dominated by the partial chunk decryptions. These can be counted by counting all *trail-subkey pairs* at each depth of the recursion. This is explained by the fact that each trail is analyzed independently of all other trails. We emphasize that summing the work done at each depth is needed to obtain an accurate estimate. Furthermore, we will (pessimistically) assume that one partial chunk decryption has cost equivalent to 1 round of the primitive (although it in fact requires just a few logic gates in the case of Speck[6]).

Claim 2. *Under the above assumptions,*

$$T_{\text{enum}} \leq \frac{R''}{R} \cdot \sum_{d=0}^{d_{\max}-1} v_d \quad FE, \tag{5.24}$$

where $R'' = 4$ in the general case. Furthermore, when the key chunks guessed are used for partial decryption of the word addition/subtraction, the complexity can be reduced by a factor of 2. In particular, $R'' = 2$ can be used in the case of Speck.

Explanation. At depth d, by Assumption 1, each trail suggests on average $2^d q_d$ candidate truncated subkeys, totalling to $n_{\text{trails}} \cdot \sum_{d=0}^{d_{\max}-1} 2^d q_d$ non-final trail-subkey pairs. For each such pair, the partial decryptions are performed for each of the two candidates for the next subkey bit and for each of the two associated state values, leading to the cost of $4/R$ FE per a non-final trail-subkey pair.

The complexity halving in the Speck case is based on the fact that, by Lemma 5, guessing i least significant bits of the (equivalent) key preceding the addition allows to check the difference for $i + 1$ least significant bits. Effectively, this means that we can replace the two checks of the two 1-bit extensions of the current guess by one. Indeed, a direct application of the general estimation would mean that, for a fixed i-bit subkey, the two checks for $(i + 1)$-bit subkey candidates would always return the same answer because the most significant bit in (truncated) addition does not affect the difference. Due to our cost estimation of 1 round of the primitive, we may

[6]Bitslice-style optimizations for reducing this crucial constant might significantly improve the attack time complexity further, compared to [Din14].

perform decryption of states only after guessing each round's subkey's most significant bit. Since this bit propagates linearly through ADD, the two subkey candidates are related by one bit flip, which has negligible cost. □

5.4.4.3 Estimating T_{enum} (Multiple-Trail)

We will model each subkey suggested by trails as sampled independently and uniformly at random. This is formalized by the following assumption.

Assumption 2. *The subkeys suggested by each trail at each depth can be modelled as random uniformly distributed subsets of all possible subkeys, sampled independently from subkeys for trails suggested by another pair.*

The validity of the assumption is not entirely obvious. It is crucial to require independence only *across different pairs*. Indeed, for one ciphertext pair, there would likely exist multiple trails of the form $\Delta C \to \Delta Z$ with prefixes equal up to some depth $d < d_{\max}$. This means that the keys suggested by these trails would be counted many times until the trails will diverge, even though they belong to a single ciphertext pair. That is why the assumption requires independence only between subkeys suggested by different pairs. In fact, the described intersections of suggested subkey sets related to a single pair of ciphertexts only reduce the number of suggested unique subkeys *per pair* and (slightly) improve the counting efficiency in practice. As we will show (see e.g. Section 5.5.3), the analysis relying on this and other used assumptions closely match experimental data.

Claim 3. *For any depth d, $0 \le d \le d_{\max}$, and any integer c, $1 \le c \ll 2^d$, let $\eta_d = n_{\text{trails}} \cdot q_d$. Under the above assumptions,*

$$T_{\text{enum}}^{c>1} \le \frac{R''}{R} \cdot \sum_{d=0}^{d_{\max}-1} 2^d \cdot \eta_d \cdot \left(1 - e^{-\eta_d} \cdot \sum_{i=0}^{c-2} \frac{\eta_d^i}{i!}\right) \quad FE, \tag{5.25}$$

where R'' is defined as in Claim 2. In particular, $R'' = 2$ in the case of Speck.

Explanation. The high-level structure of this estimation is based on counting the average total number of trail-subkey pairs processed during the procedure, similarly to Claim 2 (estimating T_{enum} for the case $c = 1$).

As was shown in Lemma 8, the average number of trail-subkey pairs at depth d for $c = 1$ is equal to $n_{\text{trails}} \cdot 2^d \cdot q_d = 2^d \cdot \eta_d$. By Assumption 2, we can model them as $2^d \cdot \eta_d$ balls thrown into 2^d bins, with each throw chosen uniformly and independently at random. Our goal is to compute the expected number of balls (trail-subkey pairs) landing in bins (subkeys) with at least c balls in each of them. □

Remark 1. Note that the expression (5.25) with $c = 1$ reduces exactly to the expression (5.24) from Claim 2, if we define the sum $\sum_{i=0}^{-1} \dots$ to be equal to zero.

5.4.5 Key Recovery Limits and Time-Data Trade-offs

In this section, we analyze the theoretical power and limits of MiF with respect to weights of involved trails or differentials. Indeed, the MiF attack has several parameters and the trade-off between the time and data complexities is not very clear. In particular, one question that we will study is how the values of the cluster/filter weights affect the time complexity (their effect on the data complexity was analyzed in Section 5.4.1.1). Here, we will assume that the key recovery procedure is perfect: given a trail or even a *differential* over k rounds, it enumerates all valid candidate subkeys without any extra overhead (for now, we set $c = 1$).

We will focus on the following MiF-like setting. Later, we will show that it can be generalized to cover other MiF variations.

An attacker uses a differential $\Delta_{\mathrm{IN}} \xrightarrow{r} \Delta_{\mathrm{OUT}}$ over r rounds with weight w and queries an $(r+k)$-round encryption of a plaintext pair (P_1, P_2) with the difference Δ_{IN}, obtaining a ciphertext pair (C_1, C_2) with a difference ΔC. The MiF tool suggests a set of valid trails of the form $\Delta C \xrightarrow{k} \Delta_{\mathrm{OUT}}$. Note that we require $k = u$ here. The attacker may run the perfect key recovery and obtain, by Assumption 1, a list of $2^{\kappa - w}$ subkeys, where κ is the size of the involved subkeys in bits (typically equal to the size of the master key). These subkeys may then be checked using trial decryptions. The question arises: should the attacker attempt the key recovery? Or, perhaps, it is better to wait for another pair?

The key insight to answering these questions lies in studying the probability that the right subkey is among the suggested subkeys *posterior to observing the output difference* ΔC. Indeed, if the suggested subkeys are not better than fully random subkey guesses, then the attack is not useful at all. On the opposite, if the suggested subkeys are g times more likely to match the right subkey, $g > 1$, then, on average, the attacker would need to test g times fewer subkeys to find the right one, effectively reducing the time complexity (more precisely, the T_{trials} component) by the factor g.

In the following, consider two differentials

$$\tau_r = \Delta_{\mathrm{IN}} \xrightarrow{r} \Delta_{\mathrm{OUT}}, \qquad \tau_k = \Delta_{\mathrm{OUT}} \xrightarrow{k} \Delta C, \tag{5.26}$$

with probabilities $\Pr[\tau_r] = p$ and $\Pr[\tau_k] = q$ respectively. Let $\tilde{p}$ be the probability of the full differential $\Delta_{\mathrm{IN}} \xrightarrow{r+k} \Delta C$.

Definition 8. Define the *gain* g of the pair (τ_r, τ_k) as

$$g = \frac{\Delta_{\mathrm{IN}} \xrightarrow{r} \Delta_{\mathrm{OUT}}}{\Delta_{\mathrm{IN}} \xrightarrow{r+k} \Delta C} = \frac{p}{\tilde{p}}. \tag{5.27}$$

Note that this definition of gain is the same as given above: the right key will be suggested by the key recovery if and only if the encrypted pair followed both differentials τ_r and τ_k; therefore, a suggested key has the probability to be the right key equal exactly to the probability that it followed the differential τ_r (the second differential is automatic since the attacker already observed ΔC).

We can upper bound the gain in two ways. First, $\tilde{p} \geq pq$ follows from the fact that the trail $\Delta_{\mathrm{IN}} \xrightarrow{r} \Delta_{\mathrm{OUT}} \xrightarrow{k} \Delta C$ is included in the differential $\Delta_{\mathrm{IN}} \xrightarrow{r+k} \Delta C$.

5.5 Attacks on 11 Rounds of Speck32

In this section, we estimate the time complexity of the MiF filtering procedure when applied to Speck32 before describing MiF attacks on Speck32 reduced to 11 rounds. Our attacks can be divided into two categories (based on the division of the rounds): Attacks using an $r + s + t$ split, which is a straightforward application of MiF, and attacks with a MiF filter of a different size. We can vary the number of r, s, t rounds depending on the number of rounds being attacked, the availability of a valid differential or to achieve other time-data trade-offs.

5.5.1 Filtering Speck32 Trails with MiF

We begin by first describing the functionality of MiF when applied to SPECK. Recall that we only need subkeys for $k = 4$ rounds to recover the full master key. Thus a straightforward application of MiF appends 4 rounds at the bottom of an r-round differential in the form of a $2 + 2$ MiF filter with $(s, t) = (2, 2)$. The operation of this filter configuration is shown in Figure 5.3. The elements in green are fixed values from the best r-round differential (trail) used in the attack. The elements in dark yellow come from the pre-computed cluster trails $\tau_s \in \mathcal{S}$. Purple elements correspond to trails $\tau_t \in \mathcal{T}$ generated by the reverse search procedure (the backward filter).

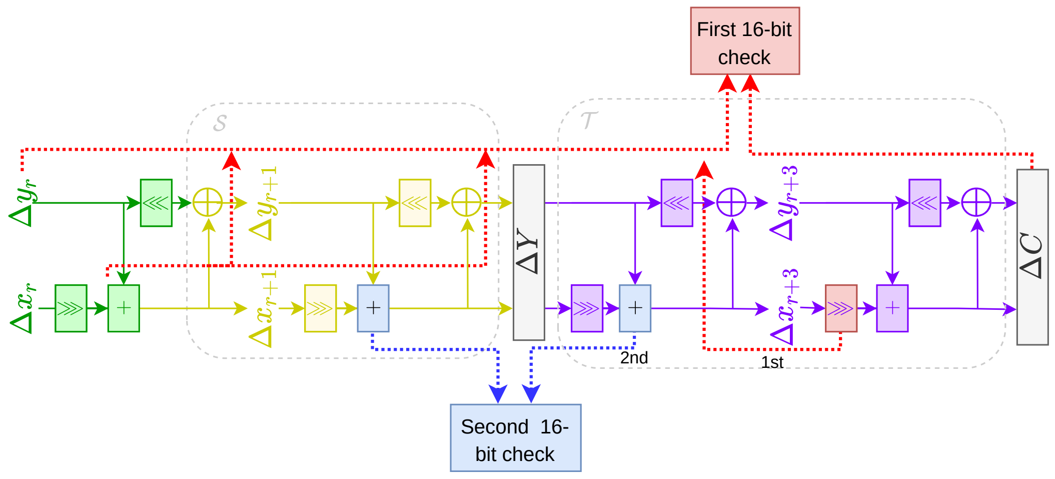

FIGURE 5.3: Operation of a $2+2$ MiF filter on the bottom four rounds of an $r+4$ round attack on SPECK$2n$. The elements in green are fixed values from an r round differential (trail). The elements in dark yellow come from a pre-computed cluster trail $\tau_s \in \mathcal{S}$. Purple elements correspond to trails $\tau_t \in \mathcal{T}$ generated by the reverse search procedure (the backward filter).

Since SPECK is a Feistel-like cipher, the match in the middle between the sets $\mathcal{S}$ and $\mathcal{T}$ can be done efficiently n bits at a time. Specifically, the first n-bit check is executed on the right branch of round $r + 3$ at the bottom (see Figure 5.3). It matches the differences generated by the ADD operation in the last round to the differences in the right branch coming from the cluster $\mathcal{S}$. This match is illustrated by the red line in Figure 5.3 (denoted “First n-bit check”). Only the trails $\tau_t \in \mathcal{T}$ that pass the first check proceed to the second n-bit check. The latter is executed on the left branch at round $r + 2$ and is illustrated by a blue line in Figure 5.3 (denoted “Second n-bit check”).

Denote by T_{b} the time complexity for checking the non-zero probability condition of Lemma 9 for a single ADD differential. Further, let T_{a} be the time complexity to generate a single output difference γ for fixed input differences α, β for one ADD operation, such that the differential $(\alpha, \beta \to \gamma)$ is of non-zero probability. The parameters $T_{\text{a}}, T_{\text{b}}$ are all measured in SPECK encryptions. Next, we give the procedure for generating the bottom 4-round trails for the (1+6+2+2)-round attack. For the sake of generality, we omit the additional round at the top in our description since it does not affect the attack complexity.

1. Encrypt $\frac{D}{2}$ chosen plaintext pairs $\{(P_1, P_2 = P_1 \oplus \Delta_{\mathrm{IN}})\}$ for $r+2+2$ rounds and collect the corresponding ciphertexts $\{(C_1, C_2)\}$. Recall that the data complexity is $D = 2 \cdot c' \cdot p^{-1} \cdot q^{-1}$ chosen plaintexts.
2. Each ciphertext pair (C_1, C_2) from Step 1 is expanded into about $2^{12.1}$ ADD differentials for the last round modular addition in time T_a SPECK-encryptions (per pair). This is the maximum number of non-zero ADD differentials for SPECK32 due to Lemma 5.
3. Of the $\frac{D}{2} \cdot 2^{12.1}$ ADD differentials from Step 2, a fraction of $\frac{|\mathcal{S}(s,w_s)|}{2^{16}}$ on average results in a match with an entry from the cluster $\mathcal{S}$. For each match, check the non-zero probability condition of Lemma 9 in time T_{b}.
4. Each ADD differential from Step 3 has a $p_z = 2^{-3.9}$ chance to be of non-zero probability (cf. Lemma 4). Therefore the total number of possible differentials surviving the $s+t$ MiF filter is $\frac{D}{2} \cdot 2^{12.1} \frac{|\mathcal{S}(s,w_s)|}{2^{16}} 2^{-3.9}$. Each one represents a candidate right pair.
5. For each trail $\tau_k = \tau_s || \tau_t$ from Step 4 execute the key-recovery procedure [7].

In Step 2, the reverse search procedure in the backward filter of SPECK32 visits at most $2^{12.1}$ ADD differentials per round. Note that starting from ciphertext differences with low Hamming weight will (significantly) reduce this number, while for random differences we shall generally see all $2^{12.1}$ transitions if there is no limit on the permissible transition weight. Since the cluster entry at the point of the match is fixed from the output difference Δ_{OUT} of the differential (Step 3), the MiF filter checks at most $\max\left(1, \frac{|\mathcal{S}(s,w_s)|}{2^n}\right)$ elements $(\Delta x_r, \Delta y_r \to \Delta x_{r+1})$ for the non-zero probability condition of Lemma 9. For example, entries in a cluster with $|\mathcal{S}| = 2^{20}$ elements will have $\frac{2^{20}}{2^{16}} = 2^4$ candidates on average, lower than the expected $2^{12.1}$ ADD transitions given in the attack procedure. Therefore the number of operations executed in the above steps is a worst-case estimate.

5.5.1.1 MiF Complexity

The complexity of the MiF filtering procedure, T_{mif} can be estimated as follows:

$$T_{\mathrm{mif}} = \underbrace{\frac{D}{2} \cdot 2^{12.1} T_a}_{\text{Steps 1, 2}} + \underbrace{\frac{D}{2} \cdot 2^{12.1} \frac{|\mathcal{S}(s,w_s)|}{2^{16}} T_{\mathrm{b}}}_{\text{Step 3}} \tag{5.28}$$

$$= D \cdot 2^{11.1} (T_a + \frac{|\mathcal{S}(s,w_s)|}{2^{16}} T_{\mathrm{b}}) \ . \tag{5.29}$$

The unit of measurement in (5.28)–(5.29) is FE. For T_{a} and T_{b} we assume that each of the three basic arithmetic operations in SPECK (addition, bitwise rotation, XOR) have the same amortized cost of 1 unit operation (UO). Thus one round of SPECK, composed of five basic operations, costs 5 UO.

For SPECK32, we estimate the cost to generate a single output difference γ for fixed input differences α, β for one ADD to be equal to 1 UO on average (3 for SPECK64), since the cDDT is able to generate a new γ at every table access, after the other parts of the word were recursively set (in the cDDT, the (α, β, γ)-differentials are processed in 8-bit chunks). The cost of checking the non-zero probability condition

[7]The reader is invited to seek the full paper for the key recovery procedure, as it is out of the scope of this thesis [Ale+22].

of Lemma 9 is estimated at 11 UO, by counting the number of operations needed to implement. With the given amortized estimations in UO units, the parameters T_a and T_b for SPECK32 reduced to R rounds are computed in terms of R-round FE as: $T_\mathrm{a} = \frac{1}{5R}$ FE and $T_\mathrm{b} = \frac{11}{5R}$ FE. The $5R$ in the denominator comes from the fact that each round has five unit operations, i.e., costs 5 UO. Note that we also assume that the cluster search can be implemented as (hash) table look-ups requiring 1 UO each. In most of our attacks, however, MiF's time complexity is not the dominating term, especially when larger clusters are in use.

5.5.2 Attacks using Splits (1+6+2+2) and (1+0+8+2)

When using MiF, we are not restricted to having $u = s + t$ rounds appended after an r-round differential. Instead, the values of r, s, t can be varied to obtain various trade-offs. One extreme would be to have all r rounds of the differential as the top half of the MiF filter, i.e., a $0 + s + t$ split with $s = r$. Note that when attacking the same number of rounds, the latter allows using longer differentials than the $r + s + t$ split. We consider two scenarios that according to our findings produced the best 11-round attacks: a (1+6+2+2) split using 6-round differentials, and a (1+0+8+2) split using 8-round differentials.

The (1+6+2+2) split follows exactly the basic structure of MiF described in Section 5.4. As for the (1+0+8+2) split, the cluster $\mathcal{S}$ will instead contain 8-round trails obtained by applying the Matsui-like search starting from the input difference Δ_IN of the 8-round differentials rather than their output difference Δ_OUT. This slightly increases the time required to pre-compute the $\mathcal{S}$ but does not affect the online phase of the attack. We only need to store information about the bottom two rounds of the 8-round trails in $\mathcal{S}$ to reconstruct the 4-round trails required for key recovery during the backward filtering procedure. Apart from using a different round configuration, the rest of the MiF filtering procedure follows the steps described in Section 5.5.1. Similarly, T_mif can be calculated based on Equations (5.28)–(5.29)

5.5.3 Results

Our strategy to find the best 11-round attacks on SPECK32 (and subsequently, other variants of SPECK) is as follows: We first identify the best differentials to be used in our attacks, some of which are listed in Table 5.6. Starting from a conservative value (usually the weight of the initial differential trail used in the attack), we increment the cluster weight w_s and compute the attack complexities for both (1+6+2+2)and (1+0+8+2) splits. Note that we always set w_t to the maximum value of 30 as to not impose any limit on the backward filtering process, thus maximising MiF efficiency. We repeat the process for all possible differentials to identify the attacks with the best time and/or data complexities. The results of our search for the (1+0+8+2) split is shown in Figure 5.4, which consists of only the best attack time complexities T for varying amounts of data D. Additionally in Table 5.3, we provide parameters for several other best attacks, including those using the (1+6+2+2) split, on 11-round SPECK32.

For 11-round SPECK32, the figure shows that using either $c = 3$ or $c = 4$ leads to optimal time complexity versus data complexity trade-offs, i.e., by *spending* more data, we get bigger gains in analysis speed. This is in contrast to using $c = 1$, which is analogous to adopting Dinur's approach [Din14], which barely sees any time complexity improvements with more data consumption. When optimized for time

No.	Split	w_s	$\lvert\mathcal{S}(s,w_s)\rvert$ ($\log_2$)	pq ($\log_2$)	c	D ($\log_2$)	T_{mif} ($\log_2$)	T_{cnt} ($\log_2$)	T_{att} ($\log_2$)	Diff. ID
1	1+6+2+2	34	19.28	-13.31	2	15.41	27.49	36.84	36.84	1
2		25	3.58	-21.30	3	24	25.13	25.09	26.11	2
3	1+0+8+2	37	21.27	-12.01	2	14.11	29.87	40.15	40.15	4
4		32	17.52	-13.48	2	15.58	25.93	34.87	34.87	5
5		24	0	-24.00	3	26.70	24.24	22.66	24.66	4

TABLE 5.3: Attacks on 11 round SPECK32: The “Diff. ID" column refers to the IDs of the differentials in Table 5.6).

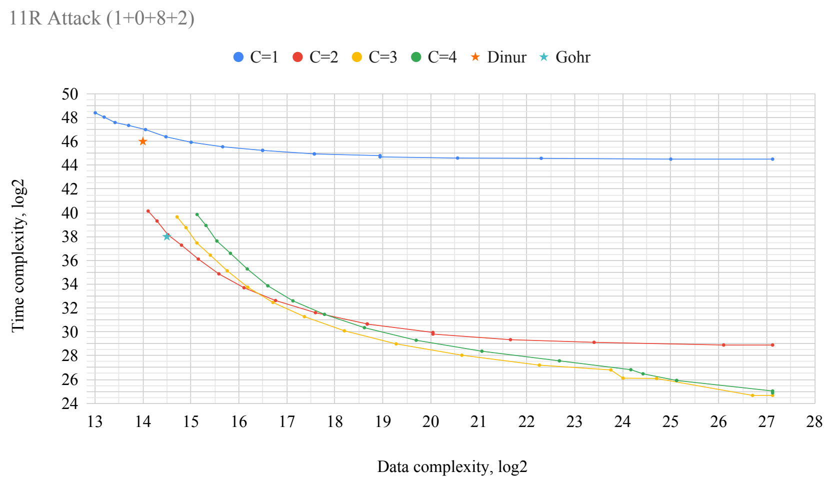

FIGURE 5.4: Time and data complexities of the best 1+0+8+2 attacks on 11-round SPECK32.

complexity, we have an (1+0+8+2) attack with $(T,D)_{11R} = (2^{24.66}, 2^{26.7})$[8] which is $2^{21.34}$ times faster than the 11-round attacks by Dinur [Din14] and $2^{13.34}$ times faster than Gohr's [Goh19]. Recall also that Gohr's attack successfully recovers 30 bits of key information 50% of the time, while our attacks recover the full master key with a success rate of around 63%.

Generally, we found that using lower cluster weights w_s lead to better attack time complexities since the resulting cluster sizes $|\mathcal{S}|$ are smaller. A smaller cluster produces fewer trails for the key recovery procedure since only a fraction of the trails in the reverse search procedure will find a match in the cluster. Fewer trails in turn reduce the total number of keys that need to be filtered. Also, a smaller cluster size allows using the simplified MiF procedure described in Section 5.5.1. We can actually push this notion to its limits by setting w_s to the weight of the corresponding trail being used in the attack, thus having $|\mathcal{S}| = 1$. For example, our fastest 11-round attack uses an input difference of $(\texttt{0x0a20}, \texttt{0x4205})$ along with an (1+0+8+2) split. This input difference corresponds to an optimal 8-round trail with probability 2^{-24}. Therefore by setting $w_s = 24$, this 8-round trail is the only one being stored in the cluster ($|\mathcal{S}| = 1$).

[8]The time complexity is less than the data complexity since it is measured in full (11-round) SPECK32 encryptions. Most of $2^{25.7}$ collected *pairs* are filtered out by MiF with the complexity of 1-round SPECK encryption.

However, going to these extremes means these attacks require the most data. When optimized for time complexity, our best attack on 11 rounds requires more (albeit still practical, $D < 2^{27}$) data than previous 11-round attacks by Dinur and Gohr, which only require 2^{14} and $2^{14.5}$ chosen plaintexts respectively. This is due to the lower efficiency q of the MiF filter, which has to be compensated by increasing the amount of data used. Thus we can reduce the data complexity for MiF attacks by using larger cluster weights, which increases both $|\mathcal{S}|$ and q. By having $w_s = 37$, we have an 11-round attack which is about 56 times faster than Dinur's attack while using a similar amount of data $(T, D)_{11R} = (2^{40.20}, 2^{14.11})$. By using $w_s = 32$, we still end up using twice as much data as Gohr, but now have an attack that is around 8 times faster $(T, D)_{11R} = (2^{35.06}, 2^{15.58})$ and with better success rate.

5.5.3.1 Experimental Verification of an 11-round Attack

The fastest 11-round attack using the (1+0+8+2) split (Attack #5 from Table 5.3) was implemented and verified in practice. We provide detailed experiment information for $c = 3$, which is the fastest variant.

The offline MiF phase generates a cluster S with the given parameters: $s = 8, w_s = 24$. Due to low cluster weight, the only trail in the cluster is the best trail $(\texttt{0x0a20},\texttt{0x4205}) \xrightarrow{8} (\texttt{0x802a},\texttt{0xd4a8})$ of weight 24. Next, $D = 2^{25.7}$ random pairs with difference $\Delta P = \Delta_{\text{IN}} = (\texttt{0x0a20},\texttt{0x4205})$ are encrypted. For each ciphertext pair, we run the simplified MiF procedure from [Ale+22] to bridge the difference $\Delta_{\text{OUT}} = (\texttt{0x802a},\texttt{0xd4a8})$ from the cluster with the ciphertext difference ΔC, and checking the resulting 2-round trail for validity (using Lemma 9). Valid trails are recorded together with the associated ciphertext pairs. Our implementation performs this procedure in several seconds. As a result, we collect $2^{14.9}$ trails, which is in line with $2^{-10.8}$ trails/pair obtained using Method 2 from Section 5.4.3.

We run the multi-trail key recovery procedure with $c = 3$ (in fact, using the secret key we could see that 5 right trails were actually suggested by MiF). Our implementation performs this procedure in less than a second, yielding the (only) right secret key. Our not fully optimized attack demonstrates significant performance improvement over the previous best attack on 11 rounds from Gohr which takes about 500 seconds [Goh19].

5.6 Attacks on 12 to 15 Rounds of Speck32

In Table 5.4 we summarize the best attacks on 12 to 15 rounds of Speck32 along with the attack parameters. For each number of rounds, we list the best attack in terms of time complexity and optimal attacks that use a similar amount of data as previous attacks in the literature. For example, the best 12-round attack using Dinur's approach [Din14] requires 2^{19} chosen plaintexts and has a time complexity of $T = 2^{51}$. In contrast, we can use slightly less data (Table 5.4, #1) for an attack that is about 34 times faster. We also have a 12-round attack that is faster than the differential-neural attack by Bao *et al.* [Bao+21] by a factor of 7.6 by only using 1.5 times more data (Table 5.4, #2). At higher rounds such as 14 and 15, the time-data trade-offs are no longer possible as we are working with almost the full codebook. Due to the restriction in data complexity, we are limited to just using $c = 1$ or 2. However, we still have 14-round and 15-round attacks that are around two to three times faster. In all cases, MiF complexity is not the dominant term and does not affect the overall analysis complexity ($T_{\text{att}} \approx T_{\text{cnt}}$).

No.	Rounds	Split	w_s	$\lvert\mathcal{S}(s,w_s)\rvert$ ($\log_2$)	pq ($\log_2$)	c	D ($\log_2$)	T_{mif} ($\log_2$)	T_{cnt} ($\log_2$)	T_{att} ($\log_2$)	Diff. ID
1	12	1+0+9+2	38	21.27	-16.17	3	18.88	32.80	45.91	45.91	8
2	12	1+7+2+2	36	15.71	-19.74	3	22.45	30.96	42.02	42.02	3
3	12	1+8+1+2	31	3.58	-27.30	4	30.42	31.42	33.54	33.84	7
4	13	1+0+10+2	43	19.38	-23.16	2	25.27	37.20	56.41	56.41	11
5	13	1+8+2+2	40	11.69	-28.01	4	31.13	36.84	50.16	50.16	6
6	14	1+9+2+2	50	17.84	-29.65	1	30.64	40.95	61.35	61.35	9
7	14	1+9+2+2	50	17.84	-29.65	2	31.75	42.05	60.99	60.99	9
8	15	1+10+2+2	55	18.18	-30.40	1	31.39	41.93	62.25	62.25	10

TABLE 5.4: Attacks on 12–15 rounds of SPECK32: The “Diff. ID" column refers to the IDs of the differentials in Table 5.6).

Experimental Verification of a 12-round Attack

The fastest attack on 12-round SPECK32-64 using the round splits $1 + 8 + 1 + 2$ (attack #3 from Table 5.4) was implemented and verified in practice. Initially, it was executed using the split $1 + 0 + 9 + 2$, however, after the inspection of the generated cluster, it became clear that the split $1 + 8 + 1 + 2$ describes it more precisely (see below). We provide detailed experiment information for $c = 4$, which is the fastest variant.

The offline MiF phase generates a cluster S with the given parameters: $s = 9, w_s = 31, \Delta_{\text{IN}} = (\texttt{0x7458}, \texttt{0xB0F8})$. Due to the low cluster weight, the generated cluster contains only 12 trails. Upon a manual inspection, it turned out that the 9-round cluster in fact consists of a single 8-round trail (namely, $(\texttt{0x7458}, \texttt{0xB0F8}) \xrightarrow{8} (\texttt{0x802A}, \texttt{0xD4A8})$ having the best possible trail weight 24), extended by 1 round in 12 different ways. The cluster has efficiency $2^{-27.30}$, catching the $2^{-3.30}$ fraction of the signal from the 8-round trail. Next, $D = 2^{29.42}$ random pairs with difference $\Delta P = \Delta_{\text{IN}}$ are encrypted. For each ciphertext pair, we run the simplified MiF procedure to bridge one of the differences ΔX from the cluster with the ciphertext difference ΔC. Our implementation performs this procedure in 11 minutes. As a result, we collect $2^{23.52}$ trails, which is in line with $2^{-5.87}$ trails/pair obtained using Method 2 from Section 5.4.3.

We run the multi-trail key recovery procedure[9] with $c = 4$. Using the secret key we could see that 7 right trails were actually suggested by MiF. The increased number of right trails was persistent across several executions. We explain this by probability increase for the cluster due to the *differential effect* of the underlying trails. This means that we could, in principle, use higher c with the same data complexity D while maintaining the target success rate above 63%. Our implementation performed this procedure in 13 minutes for $c = 4$ or in 6 minutes for $c = 7$ (on the same data set), yielding only the correct secret master key.

Our attack demonstrates significant performance improvement over the previous best attack on 12 rounds by Gohr (12 hours) [Goh19]. The illustration of the time complexity evolution of the attack for different values of c (and data complexity adapted to maintain the success rate of 63%) can be found in Figure 5.5.

[9]See full paper.

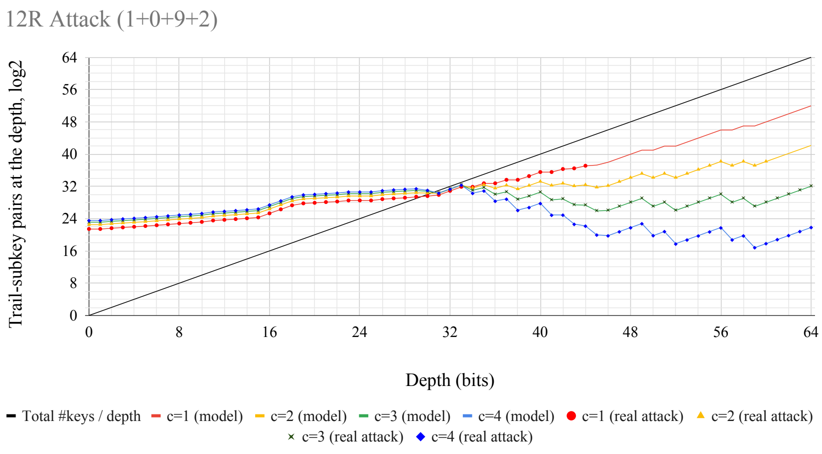

FIGURE 5.5: Time complexity analysis of an attack family on 12-round SPECK32 (see Table 5.4, #3). Lines plotted are the predicted numbers of trail-subkey pairs visited per each depth 0 . . . 64 for attacks with $c = 1, 2, 3, 4$; data points mark values collected from real attack runs, one run per each c (full attacks for $c = 3, 4$ and partial samples up to feasible depths for $c = 1, 2$).

5.7 Attacks on Speck64/128

In Table 5.5 we highlight some of our best attacks on 13, 19 and 20 rounds of SPECK64/128, all of which adopt a $1 + r + 2 + 2$ split. Contrary to intuition, our results show that using suboptimal differentials can sometimes produce better attacks due to having better trail weight distributions for key recovery e.g. 4-round trails with heavier weights in the top two rounds would suggest fewer keys than those with less. The time-data trade-offs that are possible with MiF can be clearly observed in the 13-round and 19-round attacks which both have data complexities that are well within the codebook. When using 2.5 times the data, we have a 13-round MiF attack that is around $2^{34.66}$ times faster than Dinur's approach [Din14]. By further doubling the amount of data, analysis speed is further improved by a factor of $2^{8.89}$.

No.	Rounds	Split	w_s	$\lvert\mathcal{S}(s, w_s)\rvert$ ($\log_2$)	pq ($\log_2$)	c	D ($\log_2$)	T_{mif} ($\log_2$)	T_{cnt} ($\log_2$)	T_{att} ($\log_2$)	Diff. ID
1	13	1+8+2+2	59	26.13	-29.18	2	31.28	50.96	61.34	61.34	12
2	13	1+8+2+2	56	23.71	-29.35	2	31.46	51.06	59.53	59.53	12
3	13	1+8+2+2	55	22.86	-29.44	3	32.14	51.75	51.07	52.45	12
4	19	1+14+2+2	86	26.97	-56.46	2	58.56	77.76	114.65	114.65	13
5	19	1+14+2+2	81	22.02	-57.32	6	61.03	79.80	101.08	101.08	13
6	20	1+15+2+2	92	27.98	-61.86	2	63.96	83.22	122.69	122.69	14

TABLE 5.5: Attacks on SPECK64: The "Diff. ID" column refers to the IDs of the differentials in Table 5.6).

When using around the same amount of data $D \approx 2^{61}$ as the best attacks in literature, our attack has an analysis complexity of $2^{101.08}$, which is around 2^{24} faster. Compared to SPECK32, we clearly see bigger gains when using MiF on SPECK64 because more noise (wrong trails) can be quickly discarded using the counting technique. This is due to these trails having a lower differential transition probability (Lemma 7).

When it comes to 20 rounds of Speck64/128, we face restrictions in terms of data complexity as we have almost exhausted the codebook. Thus, we are limited to using $c = 2$ in our best attack, which is still 7.3 times faster than the best 20-round attack proposed by Song *et al.* [SHY16].

5.8 Conclusions

In this paper, we proposed a new cryptanalytic technique called *Meet-in-the-Filter* (MiF). It leverages a new time-memory trade-off that reduces the data and time complexity of the analysis phase of a differential attack. MiF is especially suitable to ciphers with slow or incomplete diffusion such as the ones from the ARX family.

The main idea behind MiF is the combination of a differential trail with a deeper filtration procedure in the last rounds of the cipher. The latter employs a reverse Matsui-like search for the differences (the *filter*) that have a match (i.e., *meet*) with a set of pre-stored differences (the *cluster*) arising from the output of the differential trail. A successful match results in candidate trails that are passed on to our advanced key-recovery procedure. Since key-recovery often dominates the time complexity of the attack we modified traditional counting techniques to use bitwise depth-first search which becomes memoryless and very efficient.

We reported improved time and data complexities over the best-known attacks on Speck32, reduced to $11 - 15$ rounds. Notably our attacks significantly improve over the best attacks developed using neural networks by Gohr. We also show the best attacks on 13, 19 and 20 rounds of Speck64.

The combination of the MiF tool with the dynamic counting can be applied to other ARX ciphers which is a promising direction for the future work.

5.9 Appendixes

5.9.1 Differentials

	ID	r	Δ_{in}	Δ_{out}	Pr T	Pr D	Ref.
Speck32	1	6	`0x0211,0x0A04`	`0x850A,0x9520`	13	-	-
	2	6	`0x0A20,0x4205`	`0x8000,0x840A`	14	-	-
	3	7	`0x0A20,0x4205`	`0x850A,0x9520`	18	17.94	-
	4	8	`0x0A20,0x4205`	`0x802A,0xD4A8`	24	23.84	-
	5	8	`0x0A60,0x4205`	`0x802A,0xD4A8`	24	23.84	-
	6	8	`0x7448,0xB0F8`	`0x850A,0x9520`	24	23.95	-
	7	8	`0x7458,0xB0F8`	`0x802A,0xD4A8`	24	23.95	-
	8	9	`0x0A20,0x4205`	`0x01A8,0x530B`	31	30.37	-
	9	9	`0x8054,0xA900`	`0x0040,0x0542`	30	29.37	-
	10	10	`0x2800,0x0010`	`0x0004,0x0014`	35	30.39	[Lee+18]
	11	10	`0x7448,0xB0F8`	`0x00a8,0x520B`	37	36.30	-
Speck64	12	8	`0x00820200,0x00001202`	`0x20200000,0x01206008`	29	28.87	-
	13	14	`0x04092400,0x20040104`	`0x80008004,0x84008020`	56	55.69	-
	14	15	`0x04092400,0x20040104`	`0x808080a0,0xA08481A4`	62	60.73	-

Table 5.6: Differentials used in this paper. Where existing differentials were not available we used a SAT solver to compute them. **Pr T** is the probability of the best trail, and **Pr D** is the probability of the differential, and both are expressed as $-\log_2(Pr)$.

5.9.2 Memory Optimizations for the Multi-Trail Key Recovery Procedure

- *On-the-fly quick filtering.* In SPECK, due to a round subkey being added only to one branch, a large fraction of suggested trails does not have valid keys for decryption of the associated ciphertext pairs in accordance with the trails. Part of this filter can be implemented very efficiently using Dinur's multi-bit filters. For example, in SPECK32, 6-bit filters applied to the last round's transition keep only about 0.25 of all trails.

- *On-the-fly deep filtering.* In our attacks, the MiF backwards filter covers 2 rounds, and these 2 rounds have very high-weight transitions on average. Therefore, checking the existence of 2-round keys would allow filtering out more trails. This can be implemented by running the single-trail recursive procedure up to 2 rounds. Note that this method has negligible time overhead, in contrast to seemingly similar Dinur's initial 2- round subkey guessing. This is due to the availability of the full trail from MiF, allowing search tree cutoffs on each bit level.

- *Larger first recursion step.* The multi-trail procedure keeps a list of trails per each depth level in the recursion. These lists have quickly decreasing sizes (according to Lemma 7, the expected factor per bit of a random differential transition through ADD is $\sqrt[n]{(4/7)^{n-1}} \leq 2^{-0.75}$ for $n \geq 16$. Therefore, the total storage size expansion (compared to the size of the input list of trails) is below the sum of this geometric progression, equal to $1/(1-2^{-0.75}) \approx 2.47$. It can be effectively reduced to 1 by increasing the first recursion step's guess to several bits. This would chop off the heaviest lists of trails on the recursion path. For example, guessing 8 bits instead of 1 would replace the factor

$$1 + 2^{-0.75} + 2^{-1.5} + 2^{-2.25} + \ldots + 2^{-6} + 2^{-6.75} + \ldots \approx 2.47 \tag{5.30}$$

by

$$1 + 2^{-6} + 2^{-6.75} + \ldots \approx 1.039. \tag{5.31}$$

We remark that this step is very similar to Dinur's initial 2-round subkey guessing. However, by guessing a smaller number of bits (which is possible due to the availability of the trail) we can minimize the memory overhead without visibly affecting the time complexity.

- *Compact storage.* In our attacks on SPECK, the backwards filter covers 2 rounds. Due to the Feistel-like structure, input and output differences of 2 rounds of SPECK completely determine the intermediate differences, i.e., the full 2-round trail. Therefore, instead of storing full 4-round trails as required for the key recovery, we could initially store trails in a compressed form: the ciphertext difference ΔC and the cluster difference ΔX. The last 2 rounds of the trail can be recovered due to the aforementioned property of the Feistel structure, and the preceding rounds can be recovered from the cluster.

  Note that the (de)compression overhead on time complexity would be negligible on first depths. At a particular depth, when the size of the list of trails is sufficiently small, all the necessary auxiliary information required to minimize the time complexity can be computed and stored for subsequent computations, causing only a negligible memory overhead.

Chapter 6

Truncation of Trails

This chapter is based on the paper titled “Automated Truncation of Differential Trails and Trail Clustering in ARX” [Bir+22].

We propose a tool for automated truncation of differential trails in ciphers using modular addition, bitwise rotation, and XOR (ARX). The tool takes as input a differential trail and produces as output a set of truncated differential trails. The set represents all possible truncations of the input trail according to certain predefined rules. A linear-time algorithm for the exact computation of the differential probability of a truncated trail that follows the truncation rules is proposed. We further describe a method to merge the set of truncated trails into a compact set of non-overlapping truncated trails with associated probability and we demonstrate the application of the tool on block cipher Speck64.

My main collaboration on this chapter was on the generation of the trails, as well as the procedure for merging the trails, and clustering around suboptimal trails, namelly Sections 6.3, 6.4, and 6.5. The relaxed rules for truncation and best distinguisher for Speck64 were done by my co-authors and are presented here for the sake of completeness.

We have also investigated the effect of clustering of differential trails around a fixed input trail. The best cluster that we have found for 15 rounds has probability $2^{-55.03}$ (consisting of 389 unique output differences) which allows us to build a distinguisher using 128 times less data than the one based on just the single best trail, which has probability 2^{-62}. Moreover, we show examples for Speck64 where a cluster of trails around a suboptimal (in terms of probability) input trail results in higher overall probability compared to a cluster obtained around the best differential trail.

6.1 Introduction

Truncated differential cryptanalysis (TC) is a technique for analysing symmetric-key cryptosystems proposed in [Knu94]. It is a variant of differential cryptanalysis (DC) [BS91] and has been used successfully against a number of cryptographic algorithms such as IDEA, Skipjack and Salsa20 among others. Similarly to differential cryptanalysis, truncated cryptanalysis traces the propagation of differences through multiple rounds of a cipher. In contrast to DC, TC does not analyse full but *truncated* differences. A truncated difference is one in which only some of the bits are specified i.e. fixed to given value 0 or 1, while the rest are truncated i.e. not specified. A truncated bit is typically denoted by a $*$ symbol implying that it may take any value.

In differential cryptanalysis a sequence of differences through several rounds of a cipher is called a *differential trail* (or differential characteristic). When only the input and output differences (and not the intermediate differences) of a differential trail are specified the resulting object is called a *differential.* The analogous concepts in truncated differential cryptanalysis are *truncated differential trails* and *truncated differentials*, both being composed of *truncated* differences.

As in DC, the objective of TC is to find a truncated differential (trail) with a sufficiently high probability p over R rounds. The latter is called a *distinguisher* as it distinguishes the cipher from a random permutation, which has probability lower than p. In its most general form, the attack principle of TC is the same as in DC. Namely, the distinguisher is used to attack $R + r$ rounds for some value of r, by guessing the last r round keys, inverting the permutation and checking if the output truncated difference after R rounds matches the one computed after the inversion under the guessed key/s. The success and complexity of a TC attack crucially depends on the ability to find high probability truncated trails and differentials.

ARX (standing for Addition-Rotation-XOR) is a class of cryptographic algorithms designed using three simple arithmetic operations: modular addition, bitwise rotation and XOR. These algorithms are typically easy to describe and implement and are very efficient, especially in software. At the same time they have been notoriously difficult to analyse due to intricate dependencies between the various operations [Leu12]. As a result a significant body of research has been dedicated to the development of tools and techniques for the automated analysis of ARX.

One of the first automated techniques for constructing differential trails for ARX-based designs is due to De Cannière *et al.* [DR06]. It uses the idea of generalized bit conditions to find collisions in the hash function SHA1. A few related automated techniques have been subsequently proposed by Leurent [Leu13], Stevens [Ste13] and Mendel *et al.* [MNS11]. Similarly, all of them have been applied to hash functions. Dedicated tools for searching for differential paths in (pure) ARX ciphers have been proposed by Liu *et al.* [Liu+19], Huang *et al.* [HW19] and Biryukov *et al.* [BV14;

BVC16]. Finally, several authors have modelled the differential search problem in terms of Boolean satisfiability or mixed-integer linear programming and have proposed the use of off-the-shelf SAT or MILP solvers to find solutions in an automated way. Some results in this direction are by Mouha *et al.* [Mou+11], Fu *et al.* [Fu+16], Sun *et al.* [Sun+14; Sun+15; Sun+15] and Song *et al.* [SHY16]. The problem of clustering of differential characteristics has been researched in [AK18; SHY16; BRV14], where the authors apply SMT solvers or dedicated tools to enumerate characteristics belonging to a given differential.

In this paper we extend the set of existing tools for analysis of ARX. More specifically, we propose a new automated tool for constructing truncated differential trails for ARX from existing non-truncated ones and computing their exact probability. The main idea is to truncate every bit from the input non-truncated trail (i.e. transforming all 0 and 1 bits into a $*$), according to certain predefined propagation rules. The rules ensure that the truncated $*$ bit will propagate until the last round of the input trail so that the resulting truncated trail will remain valid and of non-zero probability for *any* assignment of the $*$. As a result, from an input trail we obtain a set of trails represented by a single truncated trail that has probability at least as high as the probability of the initial trail. In addition, we propose a method to construct a cluster of non-overlapping truncated trails composed of all possible truncations of the input (up to the propagation rules) together with its associated probability. In contrast to [AK18; SHY16; BRV14] the trails in the constructed clusters do not necessarily belong to the same differential. They have compact representation due to which the analyst is able to trace the propagation of multiple trails at the same time.

We propose two sets of truncation rules: simple rules (Section 6.3) and relaxed rules (Section 6.6). The simple rules do not consider dependencies between consecutive bits within the same round (i.e. within the same modular addition operation). Consequently, with the simple rules, truncated bits with different labels are independent from each other and can take values 0 and 1 with equal probability. In contrast, the relaxed rules are a generalization of the simple rules that is applicable also in cases in which bits within the same round are dependent on each other. In that case truncated bits with different labels are dependent on each other (often in complex ways) and may take values 0 and 1 with different probability.

Both for the simple and for the relaxed truncation rules the only assumption we rely on is the Markov assumption i.e. treating rounds as independent. In particular, we do *not* assume that individual non-truncated trails belonging to the same truncated trail have equal probability. Indeed, in general they don't and this is taken care of by the proposed tool.

The tool is useful for constructing truncated differential distinguishers which have lower data complexity than the traditional ones based on the best non-truncated trail. Its application is demonstrated on block cipher Speck64, for which we report clusters of truncated trails produced from the optimal non-truncated trails on up to 15 rounds. The latter is the highest number of rounds covered by a single trail with probability 2^{-62} higher than random 2^{-64}. For 15 rounds in particular, we report a set of 24 truncated trails, encoding 135 non-truncated trails, the top 22 of which have probability $\geq 2^{-64}$ and cumulative probability $2^{-59.05}$. The latter improves the probability of the single optimal trail by a factor of about 8 at the expense of considering multiple trails. A summary of those results is given in Table 6.1.

In the context of the existing tools mentioned earlier, the proposed tool bears similarity to the generalized conditions idea introduced in [DR06] and extended in [Leu13]. Indeed the set of truncated and fixed bits is a subset of the full set of (extended) generalized conditions. Several features set our tool apart from [DR06; Leu13]. First, by limiting ourselves to just a very small subset of the generalized conditions we are able to compute the exact probability of a single truncated trail in linear time in its length. Second, due to the same reason we are also able to transform a set of overlapping truncated trails into a set of disjoint truncated trails. The latter is critical for being able to compute the probability of a cluster of truncated trails, which on its turn is critical in estimating the data complexity of an attack. Finally, ours is a dedicated tool for finding truncated trails, while the mentioned tools have been applied in the context of collision search in hash functions.

The source code of the tool for the simple rules (Sect. 6.3,6.4 and 6.5) will be made publicly available as part of the YAARX Toolkit [Vel].

R	Δ_{in} (#)	$\#T_{\text{tr}}$	$\#T_{\text{ntr}}$	$P_{\min}$	$P_{\max}$	P_{tr}	S/N	mat.S/N	mat.P_{tr}
				($\log_2$)	($\log_2$)	($\log_2$)	($\log_2$)	($\log_2$)	($\log_2$)
5	02000012 02000002 (1)	3	20	−15	−10	−7.58	52.10	33.13	−3.70
6	00008202 00001202 (1)	6	48	−23	−15	−12.02	46.40	32.31	−6.46
6	00401042 00400240 (1)	3	20	−20	−15	−12.58	47.10	33.37	−8.92
7	92400040 10420040 (1)	3	40	−27	−21	−18.00	40.68	24.42	−13.10
7	40924000 40104200 (1)	6	48	−29	−21	−18.02	40.40	24.47	−13.05
7	C0924000 40104200 (1)	6	48	−29	−21	−18.02	40.40	24.49	−13.06
8	00008202 00001202 (2)	6	144	−42	−29	−25.00	31.83	20.94	−16.63
8	92400040 10420040 (3)	28	576	−40	−29	−23.37	31.46	21.28	−16.23
8	40924000 40104200 (3)	25	544	−41	−29	−23.40	31.51	21.28	−16.23
9	00008202 00001202 (1)	3	48	−44	−34	−30.65	27.76	20.25	−23.33
9	80240000 00040080 (1)	3	20	−39	−34	−31.58	28.10	20.87	−27.39
9	80208080 00048080 (1)	2	12	−43	−34	−32.24	28.18	20.93	−28.85
9	00802400 80000400 (1)	6	30	−46	−34	−31.58	27.51	20.86	−27.33
10	80208080 00048080 (1)	6	30	−50	−38	−35.58	23.51	20.35	−32.69
11	00000090 00000010 (1)	3	5	−45	−42	−40.75	20.93	20.19	−40.00
12	00000090 00000010 (1)	5	24	−53	−46	−43.60	15.81	11.07	−40.12
12	00008202 00001202 (1)	3	5	−49	−46	−44.75	16.93	11.59	−42.35
13	00008202 00001202 (1)	5	24	−57	−50	−47.60	11.81	10.22	−45.57
14	20200008 20200001 (1)	6	48	−64	−56	−53.02	5.40	5.06	−51.07
14	00008202 00001202 (1)	15	112 (99)	−67	−56	(−52.41)	(4.79)	4.70	−50.68
14	92400040 10420040 (1)	6	24	−63	−56	−53.60	5.81	5.70	−51.37
14	40924000 40104200 (1)	5	24	−63	−56	−53.60	5.81	4.97	−52.06
15	92400040 10420040 (1)	24	135 (22)	−74	−62	(−59.05)	(0.49)	0.37	−58.54
15	40924000 40104200 (1)	15	112 (22)	−73	−62	(−59.05)	(0.49)	0.37	−58.54
15	00040924 20040104 (1)	6	48 (16)	−70	−62	(−59.42)	(0.58)	0.50	−58.79

TABLE 6.1: Truncation of optimal trails for SPECK64. Legend: R number of rounds; Δ_{in} (#) input differences to the ADD at first round (# number of trails with such input); $\#T_{\text{tr}}$ number of truncated trails produced by the tool; $\#T_{\text{ntr}}$ number of non-truncated trails in the truncated cluster T_{tr} (in brackets are the number of trails with $\Pr \geq 2^{-64}$); $P_{\min}$ and $P_{\max}$ resp. minimum and maximum trail Pr in the set T_{ntr} ($\log_2$ scale); P_{tr} total Pr of the truncated cluster ($\log_2$ scale); $\log_2(S/N) = 64 - |\log_2(P_{\text{tr}})| - \log_2(\#T_{\text{ntr}})$; Numbers in brackets in col. $7, 8$ based on top trails in T_{ntr} with $\Pr \geq 2^{-64}$. The columns mat.S/N and mat.P_{tr} are the signal-noise and probabilities of the optimal truncation, approximated with a Matsui-search tool, whose probability limit was chosen in such a way as to make computation time feasible on a small scale server PC with a few hours of computation.

The outline of the paper is as follows. We begin with preliminaries in Sect. 6.2, followed by exposition of the rules for truncation in Sect. 6.3. In Sect. 6.4 and Sect. 6.5 is presented respectively a tool for automated truncation of differential trails in ARX and a tool for merging a set of truncated trails into a set of non-overlapping truncated trails. A set of relaxed truncation rules is described in Sect. 6.6. Results from the application of those tools to block cipher Speck64 are given in Sect. 6.7. Statistical analysis of the distinguishing advantage using truncated distinguishers is given in Sect. 6.8. In Sect. 6.9 we discuss an improved truncated distinguisher for 15 rounds of Speck64. The exposition concludes with Sect. 6.10. Notations and abbreviations are listed in Table 6.2.

Symbol	Meaning
n	Word size in bits
$\boxplus$ or ADD	Addition modulo 2^n
$\lll, \ggg$	Left, right bitwise rotation
$\wedge, \vee$	Logical AND, OR
$\overline{x}$ or $\neg x$	Logical NOT
$\oplus$	Binary exclusive-OR (XOR)
α, β, γ	n-bit XOR or truncated differences
α_i	The i-th bit of α (α_0 is LSB, α_{n-1} is MSB)
$(\alpha\beta\gamma)_i$	The i-th bits of α, β, γ as 3-bit string
$*$	Truncated bit (can be both 0 and 1)
$\tilde{*}$	Dependent truncated bit (can be both 0 and 1)
$\cdot$	Fixed bit (can be either 0 or 1)
T, τ	Truncated trail
$\mathfrak{T}, \mathfrak{t}$	Sets of truncated trails
$\#\mathfrak{T}$ or $\lvert\mathfrak{T}\rvert$	Size of the set $\mathfrak{T}$
Pr	Probability
DP	Differential probability
S/N	Signal-to-Noise Ratio

Table 6.2: Symbols and notations used in this chapter.

6.2 Preliminaries

In this section we present notations, definitions and theorems that are relevant to the subsequent parts of the paper.

By $\mathbf{xdp}^+$ is denoted the XOR differential probability (DP) of ADD and is defined below.

Definition 9. $\mathbf{xdp}^+$ is the probability with which input XOR differences α, β propagate to output XOR difference γ through the operation ADD, computed over all n-bit inputs a, b:

$$\mathbf{xdp}^+(\alpha, \beta, \gamma) = 2^{-2n}\ \#\{(a, b) : ((a \oplus \alpha) + (b \oplus \beta)) \oplus (a + b) = \gamma\}\ . \tag{6.1}$$

The following lemma provides the condition under which $\mathbf{xdp}^+$ is non-zero:

Lemma 9 (Lemma 3 [LM01])**.** *The probability* $\mathbf{xdp}^{+}(\alpha, \beta, \gamma)$ *is non-zero if:*

$$\alpha_i \oplus \beta_i \oplus \gamma_i = \begin{cases} 0 & \text{if} \quad (i = 0) \ , \\ \alpha_{i-1} & \text{if} \quad (\alpha_{i-1} = \beta_{i-1} = \gamma_{i-1}) \wedge (i > 0) \end{cases} . \tag{6.2}$$

Proof. Lemma 3 [LM01]. □

The next theorem provides a formula for the computation of $\mathbf{xdp}^{+}$.

Theorem 2 (Algorithm 2 [LM01])**.** *If* $\mathbf{xdp}^{+}(\alpha, \beta, \gamma)$ *is non-zero then its exact value is computed according to the following formula:*

$$\mathrm{xdp}^{+}(\alpha, \beta, \gamma) = 2^{-n+k+1} : \ k = \#\{i \geq 1 : \ (\alpha_{i-1} = \beta_{i-1} = \gamma_{i-1})\} \ . \tag{6.3}$$

Proof. Algorithm 2 [LM01]. □

Theorem 2 essentially states that the probability $\mathbf{xdp}^{+}$ decreases by a factor of 1/2 for every bit position i at which the three bits of the differences α_i, β_i and γ_i are *not* equal, excluding the most significant bit (MSB) (hence the +1 in the power).

A bit in a truncated differential trail can either be *fixed*, denoted by the dot symbol $\cdot$ or *truncated*, denoted by the star symbol $*$. A fixed bit has value either 0 or 1. A truncated bit can take on both values 0 and 1. More precisely, if a bit in a truncated differential trail is truncated, then the trail is valid (i.e. of non-zero probability) for both assignments of this bit.

6.3 Rules for Truncation

Truncation is performed according to three simple rules. They make truncation feasible over multiple rounds of a cipher, where the ARX operations are sequentially applied one after another. We describe those rules next, together with the rationale behind them.

Rule 4. Let (α, β, γ) be a differential through ADD. Allow at most one truncated bit in $(\alpha\beta\gamma)_i$ at all bit positions i except the least significant bit (LSB) and allow no truncated bits at the LSB:

$$(\alpha\beta\gamma)_i \in \begin{cases} \{(\cdot\cdot\cdot)\} & , \ i = 0 \ , \\ \{(\cdot\cdot\cdot), (\cdot\cdot *), (\cdot * \cdot), (* \cdot\cdot)\} & , \ n > i > 0 \end{cases} . \tag{6.4}$$

The rationale behind Rule 4 is to make truncation feasible over multiple ADD operations iterated in sequence as in an ARX algorithm. If we allow more than one truncated bit per bit position in Rule 4 then the number of $*$ bits quickly explodes in the number of rounds. Consequently it becomes infeasible to keep track of the truncated bits across multiple rounds i.e. to maintain information as to which $*$ bit at round r is related to which $*$ bit/s at round $r-1$. Note that the final goal is to end up with a truncated differential trail which results in non-zero probability non-truncated trail for *any* assignment of the $*$ bits. Finally and most importantly due to Rule 4 it is possible to efficiently (in linear time) compute the differential probability of a truncated differential through a single ADD.

Rule 5. Let (α, β, γ) be input/output differences through XOR so that $\alpha \oplus \beta = \gamma$. Allow at most one truncated bit in $(\alpha\beta)_i$ at all bit positions:

$$(\alpha\beta)_i \in \{(\cdot\cdot), (\cdot*), (*\cdot)\} : \; n > i \geq 0 \; . \tag{6.5}$$

Similarly to Rule 4, the rationale behind Rule 5 is to make it feasible to keep track of the dependency between $*$ bits over sequences of XOR operations. For example if $\alpha_i = \cdot$ and $\beta_i = *$ then the output of XOR is $\alpha_i \oplus * = *$. Thus the output star $*$ is either equal to the input star $*$ or to its negation depending on the value of α_i which is fixed. In contrast, if both input bits are truncated i.e. $\alpha_i = *$ and $\beta_i = *$ then the output is a $*$ bit that is dependent on the inputs in a (relatively) complex way.

Rule 6. Let (α, β, γ) be a truncated differential through ADD respecting Rule 4. If, at position $i - 1$, two bits are fixed and equal while the third is truncated or all three bits are fixed and equal to each other, then all bits at position i must be fixed:

$$\begin{aligned}
&((\alpha_{i-1} = \beta_{i-1} = \cdot) \wedge (\gamma_{i-1} = *)) \vee \\
&((\beta_{i-1} = \gamma_{i-1} = \cdot) \wedge (\alpha_{i-1} = *)) \vee \\
&((\alpha_{i-1} = \gamma_{i-1} = \cdot) \wedge (\beta_{i-1} = *)) \vee \\
&(\alpha_{i-1} = \beta_{i-1} = \gamma_{i-1} = \cdot) \implies (\alpha\beta\gamma)_i = (\cdot\cdot\cdot) \; .
\end{aligned} \tag{6.6}$$

Rule 6 is a consequence of the $\mathbf{xdp}^+$ non-zero condition (Lemma 9). It ensures that a non-zero probability differential (trail) remains of non-zero probability for all assignments of the $*$ bits after truncation. More specifically, if e.g. $\alpha_{i-1} = \beta_{i-1} = \cdot$ and $\gamma_{i-1} = *$ then we treat the $*$ value of γ_{i-1} as being equal to α_{i-1} in order to check that this is a valid truncation i.e. that the differential remains of non-zero Pr for both assignments of γ_{i-1}. This is the case only if $\alpha_i \oplus \beta_i \oplus \gamma_i \oplus \alpha_{i-1} = 0$, otherwise the truncation is invalid (cf. Lemma 9).

The described rules allow stars in all bit positions (even several stars per round) and in all rounds, except in the input differences. In practice however a star at a given position, for example round j, bit i, might violate one of the rules as it propagates to the last round. If that is the case, then bit (j, i) remains fixed. As a result the number of $*$ bits is relatively small. Another related consequence is that more stars appear in the last rounds since at those positions there is smaller chance to break any of the rules.

Rule 4, Rule 5 and Rule 6, when used in combination, make it possible to compute the DP of a truncated differential trail in linear time in the length of the trail.

We note that the proposed rules can be relaxed in several directions. In particular, one may relax Rule 5 by allowing two $*$ bits to enter the XOR operation. Rule 6 can be relaxed to allow a $*$ bit at a position that follows a position with equal fixed bits. Indeed we describe such a set of relaxed rules in Section 6.6. Such relaxations naturally allow to capture more signal (larger cluster of differential trails) at the expense of added complexity for keeping track of $*$ dependencies across rounds.

6.4 Differential Trail Truncation

The truncation algorithm takes a non-truncated trail as input and produces as output all its truncated variants that comply to Rules 4, 5 and 6. The input trail can be found by using one of the existing tools mentioned earlier e.g. [Liu+19; HW19; BVC16].

Denote the input trail by τ. The i-th bit at round j is denoted τ_i^j for $0 \leq i < n$, $0 \leq j < R$ and it can either be truncated or not. The algorithm explores both possibilities recursively in a depth-first search manner. Once a bit is truncated i.e. $\tau_i^j \leftarrow *$, it is propagated to the last round of the trail. The propagation through the ADD and XOR operations is performed according to Rules 4, 5, 6. Propagation through the bitwise rotation operation is done by simply rotating the $*$ bit by the corresponding rotation amount. If propagation fails for a given bit (i.e. a rule is violated), the algorithm backtracks and explores the next possibility or the next bit position. A pseudocode description of this procedure is given in the full paper.[1].

The differential probability (DP) of a truncated differential trail that follows Rules 4, 5, 6 through a single ADD operation can be computed in linear time. The procedure represents a slight modification of the one for $\mathbf{xdp}^+$ (Theorem 2) and is outlined next.

Let (α, β, γ) be a differential through ADD. In the non-truncated case the probability p of this differential decreases by a factor of $1/2$ for every bit position i at which $\alpha_{i-1} = \beta_{i-1} = \gamma_{i-1}$ does *not* hold (cf. Theorem 2). The modification of this rule concerns the cases in which there is a $*$ at some bit positions. Let $i-1$ be such a position other than the MSB i.e. $(\alpha\beta\gamma)_{i-1} \in \{(\cdot \cdot *), (\cdot * \cdot), (* \cdot \cdot)\}$ and $i \neq n$. Without loss of generality assume that $(\alpha\beta\gamma)_{i-1} = (* \cdot \cdot)$ i.e. $\alpha_{i-1} = *$. By Rule 6 it is ensured that the bits at the next position are all fixed i.e. $(\alpha\beta\gamma)_i = (\cdot\cdot\cdot)$. Two cases are possible. In the first case $\beta_{i-1} = \gamma_{i-1}$ and the probability is multiplied by 1 if $\alpha_{i-1} = \beta_{i-1}$ and by $1/2$ if $\alpha_{i-1} \neq \beta_{i-1}$ Therefore the total probability p is multiplied by $1 + 1/2 = 3/2$ in this case. In the second case $\beta_{i-1} \neq \gamma_{i-1}$ and the probability decreases by $1/2$ for both values of α_{i-1}. Thus the total probability p is multiplied by $1/2 + 1/2 = 1$ (i.e. p remains unchanged) in this case. When $i = n$ and there is a $*$ at $i-1$ (MSB), the probability p is multiplied by 2 as the value at the MSB does not change the (non-truncated) DP (cf. Theorem 2).

The differential probability of a truncated trail that follows Rules 4, 5, 6 can be computed in linear time in the length of the trail. Consider a conceptual ARX cipher with round function composed of a single ADD operation followed by a linear part composed of XOR-s and bitwise rotations. Let $\tau = (\tau^0 \ldots \tau^{R-1})$ be a truncated differential trail over R rounds such that the differential transition at round $0 \leq j < R$ represents input/output differences to ADD i.e. $\tau^j = (\alpha^j, \beta^j, \gamma^j)$. The DP of a single non-truncated differential $(\alpha^j, \beta^j, \gamma^j)$ at round j can be computed bitwise with bit i conditioned on bit $i-1$ using Theorem 2 as follows:

$$\mathbf{xdp}^+(\alpha^j, \beta^j, \gamma^j) = p_0^j \prod_{i=1}^{n-1} p_i^j : \; p_i^j = \mathsf{DP}[(\alpha\beta\gamma)_i^j \mid (\alpha\beta\gamma)_{i-1}^j] \,, \tag{6.7}$$

where $p_0^j = \mathsf{DP}[(\alpha\beta\gamma)_0^j]$. Therefore, under the Markov assumption, the probability of the trail τ is computed as $\mathsf{DP}[\tau] = \prod_{j=0}^{R-1} p_0^j \prod_{i=1}^{n-1} p_i^j$.

Notice that the probabilities p_i^j in the expression for $\mathsf{DP}[\tau]$ can be computed in any

[1]Full pseudocode for the simple rules is available as part of the YAARX Toolkit at `https://github.com/vesselinux/yaarx/blob/master/txt/arxtrunc.pdf` .

order (for a fixed τ). When computing the DP of a truncated trail τ, we are ordering the terms p_i^j by the dependency of the $*$ bits in consecutive rounds. Then we compute each term for both possible values 0 and 1 of the $*$ bit denoted resp. $(p_i^j)_0$ and $(p_i^j)_1$ and we sum the two products.

For example suppose that $\tau_r^k = *$ for some round k and bit r and that this $*$ bit propagates to subsequent rounds, up to the final one, at positions $\tau_s^{k+1}, \ldots, \tau_t^{R-1}$. Suppose also that these are the only $*$ bits in τ. The truncated DP of τ then is computed as:

$$\prod_{\substack{(i,j)\notin \\ \{(k,r),(k+1,s)\ldots(R-1,t)\}}} p_i^j \Big((p_r^k)_0 (p_s^{k+1})_0 \ldots (p_t^{R-1})_0 + (p_r^k)_1 (p_s^{k+1})_1 \ldots (p_t^{R-1})_1\Big) \tag{6.8}$$

In equation (6.8), we are essentially splitting the trail τ into bitwise subtrails, where each subtrail contains $*$ bits that are directly dependent on each other. Note that Rules 4, 5, 6 ensure that there are no dependencies between the $*$ bits belonging to different subtrails. In other words, if a $*$ bit is part of a given subtrail, then it can not be part of other subtrails.

An example 6 round truncated trail on Speck32 generated with Algorithm 22 is shown in Appendix 6.11.4, Table 6.3. It has probability $2^{-11.16}$ and has been produced from an optimal 6 round trail on Speck32 with probability 2^{-13}. The shown truncated trail has 4 independent $*$ bits and therefore encodes 16 non-truncated trails. The dependency between the $*$ bits is shown in the equivalent label representation in the third column of the table. The 4 independent truncated bits are denoted by the labels `a,b,c,d`. Another example of a strongly truncated trail produced from a suboptimal trail for Speck32 and encoding 512 non-truncated trails is shown in Appendix 6.11.4 Table 6.4.

6.5 Merging of Truncated Trails

In the general case an input non-truncated trail may have more than one possible truncation. Therefore the set of all possible (up to Rules 4, 5, 6) truncated trails produced from a given non-truncated trail by Algorithm 22 may contain duplicate non-truncated trails. In this section we describe a method to transform this set into a set of truncated trails that are disjoint i.e. do not contain any duplicates.

As described earlier, a truncated difference (TD) α represents a set of non-truncated differences defined by the $*$ bit positions in its truncated representation. We say that two TD α and α' are *disjoint*, denoted as $\alpha \cap \alpha' = \emptyset$, if their corresponding sets are disjoint i.e. if they do not have any common non-truncated differences. Note that α and α' are disjoint if there is at least one bit that is fixed and of opposite value in each TD i.e. $\exists i : \ 0 \le i < n : \ (\alpha_i = \cdot) \wedge (\alpha'_i = \cdot) \wedge (\alpha_i \ne \alpha'_i)$.

If the set represented by α is fully contained in the set represented by α' then we say that α is a *subset* of α' denoted as $\alpha \subset \alpha'$. If α and α' are not subsets of each other and are not disjoint i.e. if $(\alpha \not\subset \alpha') \wedge (\alpha \not\supset \alpha')$ and $\alpha \cap \alpha' \ne \emptyset$ then we say that α and α' are *partially overlapping* (PO). The latter implies that some, but strictly not all, differences that are in α are also in α' and vice versa. Note that if α and α' are PO then there exists at least one bit position i at which $(\alpha_i = *) \wedge (\alpha'_i = \cdot)$ and there exists at least one bit position j at which $(\alpha_j = \cdot) \wedge (\alpha'_j = *)$, where clearly $i \ne j$.

The terms *disjoint*, *subset* and *partially overlapping* have analogous meaning for the cases of truncated differentials through ADD (α, β, γ) and of truncated trails $\tau = ((\alpha^0, \beta^0, \gamma^0), (\alpha^1, \beta^1, \gamma^1) \ldots)$ composed of ADD truncated differentials.

The merging algorithm takes as input a set $\mathfrak{T}$ of truncated trails T that are all pairwise disjoint and a truncated trail τ to be merged with $\mathfrak{T}$. The output is an updated set $\mathfrak{T}$ composed of disjoint truncated trails and containing all (non-truncated) trails from τ that were not initially in $\mathfrak{T}$. For each truncated trail T in $\mathfrak{T}$, the algorithm checks three cases. If τ is already in T i.e. $\tau \subset \mathrm{T}$ then output $\mathfrak{T}$ and terminate. If τ and T are disjoint i.e. $\tau \cap \mathrm{T} = \emptyset$ then move on to the next trail in $\mathfrak{T}$ or add τ to $\mathfrak{T}$ if all trails in $\mathfrak{T}$ have been processed. Finally, if T is a subset of τ i.e. $\mathrm{T} \subset \tau$ or if T and τ are partially overlapping, then split τ into a set of truncated trails $\mathfrak{t}$ (explained below). The set $\mathfrak{t}$ is such that all its elements are pairwise disjoint and each trail from $\mathfrak{t}$ is disjoint to T. With this the procedure is finished for the trail T and moves on to the next trail in $\mathfrak{T}$ where it performs the same steps for each trail in the set $\mathfrak{t}$. The process terminates either when the set $\mathfrak{t}$ becomes empty (i.e. the initial trail τ has been fully absorbed into $\mathfrak{T}$) or when all trails from $\mathfrak{T}$ have been processed, in which case the set $\mathfrak{t}$ is added to $\mathfrak{T}$. In both cases the updated set $\mathfrak{T}$ is returned. Pseudocode description of this procedure is given in Algorithm 23.

A step that needs clarification in the described procedure is how the trail τ is split into pairwise disjoint trails $\mathfrak{t}$ that are also disjoint to T. Let $\mathrm{T} = ((\alpha^0, \beta^0, \gamma^0), (\alpha^1, \beta^1, \gamma^1) \ldots)$ and $\tau = ((a^0, b^0, c^0), (a^1, b^1, c^1) \ldots)$. By design we know that either $\mathrm{T} \subset \tau$ or T, τ: PO. In either case there must be at least one bit position i and round j for which the bit in T is fixed and the same bit in τ is truncated. Let α_i^j be one such bit i.e. $(\alpha_i^j = \cdot) \wedge (a_i^j = *)$. We construct a new trail τ' by setting a_i^j to the opposite value of α_i^j i.e. $a_i^j = 1 \oplus \alpha_i^j$. Note that this makes τ' disjoint from T since it differs in one fixed bit. We add τ' to $\mathfrak{t}$ and we discard the original trail τ. By doing so we don't lose any information since τ for $a_i^j = \alpha_i^j$ is already in T and τ for the opposite value $a_i^j = 1 \oplus \alpha_i^j$ is in $\mathfrak{t}$. If T and τ happen to differ also in another bit, say $(\beta_l^k = \cdot) \wedge (b_l^k = *)$ then we set a_i^j to the value in T: $a_i^j = \alpha_i^j$ and we set b_l^k to the negated value in T: $b_l^k = 1 \oplus \beta_l^k$. We add this new trail τ'' to $\mathfrak{t}$. Now $\mathfrak{t}$ contains τ' and τ'' which are pairwise disjoint since they differ in a_i^j. At the same time they are also disjoint from T since they differ from T respectively in a_i^j and b_l^k. This procedure is executed iteratively for all positions in which τ is fixed and T is truncated.

In Algorithm 23 there are two nested for-loops – the outer over $\mathfrak{T}$, the inner over $\mathfrak{t}$, where the size of $\mathfrak{t}$ is at most the number of non-truncated trails in τ. Therefore the complexity of the algorithm is quadratic in $\max(\#\mathfrak{T}, \#\mathfrak{t})$, where $\#\mathfrak{T}$ is the number of truncated trails in $\mathfrak{T}$ and $\#\mathfrak{t}$ is the number of non-truncated trails in τ.

6.6 Relaxed Rules

It is possible to generalize the truncation rules from Section 6.3 by allowing truncations with dependent truncated bits $\tilde{*}$, i.e. bits whose value depends (non-linearly) on previous bits' assignments and for which Lipmaa-Moriai conditions are automatically satisfied. Those relaxed rules are out of the scope of this thesis, and can be found in the full paper [Bir+22].

6.7 Application to SPECK64

Speck is a family of lightweight block ciphers proposed in [Bea+13]. The family has five members corresponding resp. to the block sizes $32, 48, 64, 96$ and 128 bits and denoted by SpeckN, where $N/2$ is the word size in bits. In the remaining part of this exposition we shall be concerned with Speck64 i.e. the variant with 32-bit words. Speck64 has two variants: 96-bit key and 26 rounds, and 128-bit key and 27 rounds.

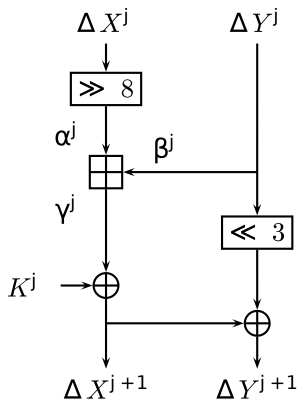

Figure 6.1: The round function of Speck64 with differential inputs.

Denote by X^j and Y^j the left and right 32-bit input words to the j-th round of Speck64 $(0 \leq j \leq R)$ and by K^j the 32-bit round key applied at round j $(0 \leq j < R)$. The output X^{j+1}, Y^{j+1} from round j is computed as follows:

$$X^{j+1} = ((X^j \ggg 8) \boxplus Y^j) \oplus K^j \quad , \tag{6.9}$$

$$Y^{j+1} = (Y^j \lll 3) \oplus X^{j+1} \quad , \tag{6.10}$$

where $\boxplus$ denotes addition modulo 2^n for $n = N/2 = 32$. The round function of Speck64 with differential inputs is shown in Fig. 6.1.

We have applied the tool for automated truncation (Algorithm 22) and merging of truncated trails (Algorithm 23) to the optimal (non-truncated) differential trails of Speck64 for up to 15 rounds. The results are shown in Table 6.1. Explanation and analysis of the data in the table follows.

The first column of Table 6.1 gives the number of rounds R. The second column shows the input difference Δ_{in} of the input optimal trail followed by the number of such trails with this input difference (in brackets). From the input trail/s [2], a set of $\#T_{\text{tr}}$ non-overlapping truncated trails (column 3) is computed by applying all possible truncations (up to Rules 4, 5, 6) and merging them. The set T_{tr} contains $\#T_{\text{ntr}}$ number of distinct non-truncated trails (column 4) with probabilities ranging from $P_{\max}$ and $P_{\min}$ (columns $5, 6$). The total probability of the truncated set T_{tr} is P_{tr} (column 7). The last column of Table 6.1 shows the $\log_2$ of the signal-to-noise ratio (S/N) computed as $\log_2(S/N) = 64 - |\log_2(P_{\text{tr}})| - \log_2(\#T_{\text{ntr}})$ (we elaborate further on this parameter below). Numbers in brackets in the $\#T_{\text{ntr}}$ column show the number of trails in the set T_{ntr} that have $\Pr \geq 2^{-64}$ (the probability of a random output difference). Correspondingly, the numbers in brackets in the last two columns are based on this subset of trails of T_{ntr} (as opposed to the full set T_{ntr}).

[2]There can be more than one input trail, provided that they share the same input difference

The S/N ratio shown in the last column of Table 6.1 is the ratio between the probability of the truncated set distinguisher P_{tr} and the probability of choosing at random a ciphertext difference that belongs to the set T_{ntr}: $\#T_{\text{ntr}} \cdot 2^{-64}$. Note that all ciphertext differences composing the distinguisher are unique. To ensure this, trails with the same output difference are "merged" in one and their probabilities are summed. The S/N ratio is an indicator of the strength of the truncated differential set distinguisher. In particular, when $S/N > 1$ the distinguisher can be used to distinguish the cipher from a random permutation.

The data in Table 6.1 indicates that the probability of the truncated differential set P_{tr} is strictly higher than the probability of the underlying optimal non-truncated trail $P_{\max}$. Consequently a truncated differential set distinguisher built around the optimal non-truncated trail is better than just single optimal trail in most cases (see next) in terms of data complexity.

The above conclusion has to be applied with caution. In particular, one has to be careful when the probability of the truncated distinguisher approaches the probability of the random event i.e. when $\Pr_{\text{tr}} \approx \#T_{\text{ntr}} \cdot 2^{-64}$ as then the S/N can easily drop below 1. This indeed happens in the case of the 15 round truncated distinguishers for Speck64 (see Table 6.1). If the full truncated sets are used as distinguishers in those cases, the corresponding three S/N ratios are $2^{-1.42}$, $2^{-1.21}$ and $2^{-0.60}$ all of which are below 1. To increase them, one has to consider only those non-truncated trails from the sets that have probability $\geq 2^{-64}$. For the three 15 round distinguishers from the table, these are the top 22, 22 and 16 trails respectively (as indicated by the numbers in brackets in the T_{ntr} column). By discarding all trails with $\Pr < 2^{-64}$ in those cases, the S/N ratios are increased respectively to $2^{0.49}$, $2^{0.49}$ and $2^{0.59}$ as shown in the table.

Another observation from the data in Table 6.1 is that some input trails have higher truncation rate (more number of truncated bits) than others. The reason for this is the specific structure of the trails with respect to Rules 4, 5 and 6. More specifically, for some trails the rules are contradicted in smaller number of bit positions (higher truncation rate) than in others.

For example from the input trail on 11 rounds, 3 truncated trails are produced containing (only) 5 non-truncated ones. At the same time the first input trail on 12 rounds (starting with the same input difference as the 11 round one) is truncated into a set of 5 truncated trails containing 24 non-truncated. Upon inspection we could see that the trail on 11 rounds has a very *thick* (i.e. low probability) transition at round 5 (counting from 0) that costs 2^{-13}, followed by *thin* (i.e. high probability) transitions until the end. So one explanation of the mentioned effect is that the thick transition breaks all rules up to round 5, while the following thin transitions don't offer many options for truncation. Interestingly the trail on 12 rounds is an extension of the one on 11 and the better truncation rate there is due to the extra round added at the end.

In the following section we provide a more detailed statistical analysis of the distinguishing advantage of distinguishers built from clusters of differential trails.

6.8 Distinguishing Advantage

Distinguishing from Random In this section we provide a probabilistic model for distinguishers for Speck built from clusters of trails. In this setting, the attacker does some pre-processing by analysing the cipher (i.e. collects trails, computes their differential weights and clusters them by weights) and then queries an oracle black-box, that can either be a `speck-box`, which returns SPECK encryptions for a uniformly chosen key, or a `random-box`, which returns random values.

Assume that in the pre-processing phase the attacker has collected **disjoint** clusters of trails $\{C_i\}_{i=0,\dots,l}$ where C_i has weight w_i (i.e. probability 2^{-w_i}) so that a random trail belongs to it with probability p_{C_i}.

Thus, if we're in

- `random-box` then $p_{C_i} = \frac{|C_i|}{2^{64}}$;
- `speck-box` then $p_{C_i} = |C_i| \cdot 2^{-w_i} = |C_i| \cdot 2^{-(w_0+i)}, i = 0, 1, \dots, l$, where w_0 is the weight of the best differential trail.[3] We consider trails with weight increasing from the optimal one. Thus we express trail weights w_i as $w_i = w_0 + i$ where w_0 is the optimal weight and i ranges from 0 to a given bound.

The probability p to hit at least 1 ciphertext in a collection of clusters after 2^N queries to the oracle is then equal to

$$p = 1 - \Pr(\text{none of the ciphertexts is in any cluster}) = 1 - (1 - \sum_{i=0}^{l} p_{C_i})^{2^N}$$

By approximating $\left(1 - \frac{1}{x}\right)^n \approx e^{-n/x}$, we can rewrite this probability as

$$1 - \left(1 - \sum_{i=0}^{l} p_{C_i}\right)^{2^N} = 1 - \left(\frac{1}{e}\right)^{2^{N+\log_2\left(\sum_{i=0}^{l} p_{C_i}\right)}} = 1 - \left(\frac{1}{e}\right)^{2^{N+k}}$$

Where for the `speck-box` we have

$$k_{speck} \doteq \log_2\left(\sum_{i=0}^{l} p_{C_i}\right) = -w_0 + \log_2\left(\sum_{i=0}^{l} |C_i| \cdot 2^{-i}\right)$$

while for the `random-box` we have

$$k_{rand} \doteq \log_2\left(\sum_{i=0}^{l} p_{C_i}\right) = \log_2\left(\sum_{i=0}^{l} |C_i|\right) - 64$$

It follows that if the attacker wants to hit with probability p a ciphertext in any cluster, he then needs to make 2^N queries, where N is equal to

$$N = \log_2\left(-\log(1-p)\right) - k$$

and k is either k_{speck} or k_{rand} depending on his guess for the oracle box.

Note that both models `random-box` and `speck-box` are similar and differ only in their terms k_*: in fact, the more these two values differ, the easier would be to distinguish

[3] In practical attacks the differential effect would increase these probabilities and make the distinguisher better.

points belonging to one model or the other. We then define

$$k_{speck} - k_{rand} \;=\; -w_0 + \log_2\left(\sum_{i=0}^{l} |C_i| \cdot 2^{-i}\right) - \log_2\left(\sum_{i=0}^{l} |C_i|\right) + 64 = S/N$$

(which corresponds to the S/N definition we introduced in previous Sections) and the higher this value is, the better we distinguish the two boxes. We assume that there exists at least one trail of weight less than 64 for the reduced Speck64 (i.e. $w_0 < 64$) and in order for the distinguisher to work we require $S/N > 1$ thus:

$$S/N_l \;\doteq\; 64 - w_0 + \log_2\left(\sum_{i=0}^{l} |C_i| \cdot 2^{-i}\right) - \log_2\left(\sum_{i=0}^{l} |C_i|\right) > 1$$

From this inequality and given the histogram of cluster sizes $|C_i|$ we can derive the optimal l up to which we can grow our collection of signal ciphertexts. The main criteria for the attacker is to minimize the amount of data for the distinguisher, i.e. minimizing $N = \log_2(-\log(1-p)) - k_{speck}$ (which is equivalent to maximizing the collection weight l), while keeping $S/N = k_{\text{speck}} - k_{\text{rand}} > 1$. The larger the gap $64 - w_0$ the higher l the attacker can afford.

Statistical Distinguisher. Here we provide a statistical test to distinguish with a certain confidence level α if the queried box is the `random-box` or the `speck-box`. We can model our experiments using geometric distribution of parameter p, i.e. $X_{rand}, X_{speck} \approx Geo(p)$. [4]

The two statistical alternative hypothesis can be then formulated as follows:

- H_0: $p = p_{speck} = \sum_i |C_i| \cdot 2^{-w_i}$, i.e. encryptions come from the `speck-box`.
- H_1: $p = p_{rand} = \sum_i |C_i| \cdot 2^{-64}$, i.e. encryptions come from the `random-box`.

Given a certain confidence level α (e.g. $\alpha = 0.05$) we want to compute a threshold t_α so that if the first matching ciphertext is found after $X_* = 2^N$ encryptions, we accept H_0 if $2^N \leq t_\alpha$, otherwise if $2^N > t_\alpha$ we accept H_1. We then have the following

$$\begin{aligned}\Pr(\text{Reject } H_0 \,|\, H_0) \;&=\; \Pr(X > t_\alpha \,|\, p = p_{speck}) \;=\; \sum_{k=t_\alpha+1}^{\infty} (1-p_{speck})^{k-1} p_{speck} \\ &= (1-p_{speck})^{t_\alpha} p_{speck} \cdot \sum_{l=0}^{\infty} (1-p_{speck})^{l} \;=\; (1-p_{speck})^{t_\alpha}\end{aligned}$$

By requiring $\Pr(\text{Reject } H_0 \,|\, H_0) \leq \alpha$, we have at least

$$t_\alpha = \left\lceil \frac{\ln \alpha}{\ln(1 - p_{speck})} \right\rceil$$

Thus, given such threshold t_α, the probability of accepting H_0 while being in H_1 would then be equal to

$$\Pr(\text{Accept } H_0 \,|\, H_1) \;=\; \Pr(X \leq t_\alpha \,|\, p = p_{rand}) \;=\; 1 - (1 - p_{rand})^{t_\alpha}$$

[4]For Speck, this is a consequence of the assumed Markov assumption.

Best distinguishing confidence level α. We are interested in achieving the highest distinguishing power possible within the statistical model outlined above. In practice, for a given distinguisher with probabilities p_{rand}, p_{speck}, we would like to choose a confidence level α which maximizes

$$f(\alpha) = \Pr(\text{Accept } H_0 \mid H_0) - \Pr(\text{Accept } H_0 \mid H_1)$$

where we assume at least $f(\alpha) > 0$. By expanding this definition, we get

$$f(\alpha) = (1 - p_{rand})^{t_\alpha} - (1 - p_{speck})^{t_\alpha} = (1 - p_{rand})^{\frac{\ln \alpha}{\ln(1-p_{speck})}} - \alpha = \alpha^c - \alpha$$

where $c = \frac{\ln(1-p_{rand})}{\ln(1-p_{speck})}$. So, a solution for $f'(\alpha) = 0$, would then be $\alpha = (\frac{1}{c})^{\frac{1}{c-1}}$ and this is a local maximum if $f''(\alpha) = c \cdot (c-1)\alpha^{c-2} < 0$. Thus, since $c > 0$, $\alpha = c^{\frac{1}{1-c}}$ is a local maximum when $c < 1$ or, equivalently, when $p_{speck} > p_{rand}$.

Experimental verification. We have experimentally verified the above probabilistic and statistical model. More precisely, we have run a distinguishing attack on Speck32 reduced to 9 rounds. We have collected a cluster of differentials with the input difference (`0211, 0a04`) with a cumulative probability of at least $2^{-25.4}$. These were gathered by first finding optimal full trails until weight 32, of which there were 30 unique input/output pairs, and then calculating all the possible trails until weight 40 on these input/output pairs to accommodate for the differential effect. This resulted in a S/N ratio of 1.7. Then, using the formula from the previous section we can calculate the highest distinguishing α, which in this case is $\alpha = 0.1825$, which results in the threshold $t_\alpha = 2^{26.16}$. Using t_α we can also calculate the probability of false positives for the random permutation box given t_α samples, i.e. $\Pr(\text{Accept } H_0 \mid H_1)$, which is equal to 0.408. The distinguishing gap is $0.8175 - 0.408 = 0.4095$ which is clearly significant.

In our experiment we have used two boxes (Speck32 with 9 rounds and the random permutation). For the test with the Speck box, we encrypt t_α random input pairs with the fixed input difference, and record a success if any of the output differences is equal to one specified by our differentials. For the random box we use Speckey32 with 40 rounds (as that should emulate a random permutation), and similarly we encrypt t_α pairs of samples with the same input difference, and record a success if any of the two differences are in our set of output differences.

We ran both the Speck32 and the random experiment 1000 times, and received 892 successes for Speck32 (hinting at an even higher real differential probability), and 404 successes for the random variant, verifying our statistical model.

6.9 Best Distinguisher for Speck64

In this Section we will discuss the best distinguisher we have found for 15 rounds of Speck64. Aiming at finding the most suitable one, we considered 4 optimal trails of weight -62 found with Matsui's search with input differences $\Delta_0 = (\Delta x_0, \Delta y_0)$ equal to (`40004092, 10420040`), (`04092400, 20040104`), (`92400040, 40104200`), (`924000c0, 40104200`), respectively.

Given an optimal 15 rounds trail, we split it in $n + k$ rounds; by iteratively setting $k = 3, 4, 5$ we compute the best feasible approximation of the differential probability for the first n rounds while maximizing the S/N ratio obtained from freely varying the difference transitions in the last k rounds.

Since computing the differential probability over $n = 12, 11, 10$ rounds (depending on the value of k set) quickly becomes prohibitive as the minimum trail weight limit decreases, we split the first n rounds in two chunks of j and $n-j$ rounds, respectively. Hence, by iteratively setting $j = 3, \dots, n-3$ we independently compute the two differential probabilities of these two chunks and we select the best index j so that $\Pr(\Delta_0 \to \Delta_j) + \Pr(\Delta_j \to \Delta_n)$ is minimum.

In order to approximate the differential probability of the two sub-trails $\Delta_0 \to \Delta_j$ and $\Delta_j \to \Delta_n$, we use an SMT solver to find all trails with such input/output differences and weight exceeding at most -25 with respect to their optimal weight.

The trail that performed better within this framework is the one we report in Appendix 6.11.6 Table 6.6 with parameters $k = 3$, $n = 12$ and $j = 3$. More precisely, for the differential $\Delta_0 \to \Delta_3$ we found 6 trails of total probability -10.954, while for $\Delta_3 \to \Delta_{12}$ we found 21022 trails of total probability -37.418. Thus

$$\Pr(\Delta_0 \to \Delta_{12}) \geq 2^{-48.372}$$

We then proceed by collecting all possible $k = 3$ rounds trails with input difference equal to Δ_{12} and weight less equal -12, as long as the total S/N remains greater than 0: for $\Delta_{12} = (\texttt{00080000}, \texttt{00080000})$ we obtained 389 unique Δ_{15} of weight at least -16 with total weight -6.731 and $S/N = 0.361$. We further slightly improve the total weight to -6.657 by computing the differential probability of each 3 round trail found $\Delta_{12} \to \Delta_{15}$.

This, gives us a distinguisher of probability $p_{speck} = 2^{-48.372-6.657} = 2^{-55.029}$ consisting of 389 unique ciphertexts.

Given that $p_{rand} = \frac{389}{2^{64}} = 2^{-55.396}$, in light of the previous section, we obtain the best confidence level $\alpha = c^{\frac{1}{1-c}} = 0.322$ with $c = \frac{\ln(1-p_{rand})}{\ln(1-p_{speck})} = 0.775$ which in turn correspond to a distinguishing threshold $t_\alpha = 2^{55.576}$ and distinguishing gap of 0.094, small but non-negligible.

Best cluster around sub-optimal trail. It is natural to use the best trail as a starting point for a trail cluster. However one may wonder if it always produces the cluster with the highest total probability for a given S/N ratio. Interestingly the answer is "no". In some cases the cluster around sub-optimal trail will have higher distinguishing power than the one starting from the best trail. In Appendix 6.11.3 we show this behaviour for clusters collected for SPECK 64 reduced to 11 and 14 rounds. The plots showing this behaviour are in Figures 6.2 and 6.3. For 11 rounds there is one best trail and there are numerous sub-optimal trails with better clusters. For 14 rounds there are three best trails and there are two sub-optimal trails which are better than two of them and very close to the very best cluster. For 15 rounds sub-optimal trails always have weaker clusters but the four available best trails differ significantly. This fact has helped us to build the best distinguisher for 15 rounds SPECK 64 described above.

These results were obtained by exploring sub-optimal trails up to certain weight bound beyond the best trail. This analysis shows that when deciding on a number of rounds of a cipher it might be important to consider not only the best differential, but the best differential cluster.

6.10 Conclusion

In this paper we described a new tool for the automated truncation of differential trails in ARX. The tool generates all possible truncations of an input non-truncated trail (up to certain pre-defined rules) and outputs a set of non-overlapping truncated trails with associated probability. The latter is strictly greater than the probability of the input trail. The proposed tool is useful for constructing truncated differential distinguishers which have lower data complexity than the traditional ones based on the best non-truncated differential. Interestingly, in some cases differential cluster around sub-optimal trail gives better resulting distinguisher then when starting from the best trail.

The application of the tool was demonstrated on block cipher SPECK64. More specifically, truncated differential set distinguishers based on the optimal trail/s on up to 15 (out of 24) rounds were reported. A natural future direction is the application of the tool to other ARX algorithms. Beside other ciphers, the tool could potentially be used in the area of ARX-based hash functions and sponge permutations. In particular, it may be worth exploring its use in an initial pre-processing phase that would facilitate the subsequent application of advanced collision search tools such as e.g. [Ste13; MNS11; Leu13; LP20].

6.11 Appendix

6.11.1 Differential Trail Truncation Algorithm

Algorithm 22 Truncation of Differential Trails in ARX

```
 Input:  $i:\ 0 \le i < n$: bit position; $j:\ 1 \le j < R$: round position
$\tau$: non-truncated trail on $R$ rounds, where $\tau_i^j$ is the $i$-th bit at round $j$
 Output:  $\{\tau\}$: all truncations of $\tau$ that follow Rule 1, Rule 2 and Rule 3
procedure truncate_trail
if $j < R$ then
    for truncate = true, false do
        // truncate bit $\tau_i^j$
        if truncate = true and $\tau_i^j \neq *$ then
            Truncate bit $\tau_i^j \leftarrow *$ and propagate to rounds $j+1, \ldots, R-1$
            if Rules 1,2 and 3 are not violated for any round then
                Update $\tau$ with $\tau^j, \tau^{j+1} \ldots \tau_j^{R-1}$
                Call truncate_trail for next bit $i+1$ or next round $j+1$
            end if
        end if
        // do not truncate $\tau_i^j$: move to next bit
        if truncate = false then
            Call truncate_trail for next bit $i+1$ or next round $j+1$
        end if
    end for
else
    // Last round: return a truncated version of $\tau$
    return $\tau$
end if
```

6.11.2 Trail Absorption Algorithm

Algorithm 23 Absorb a new TD trail $\boldsymbol{\tau}$ into exisiting set of trails $\mathfrak{T}$

```
**Input:** $\mathfrak{T}$: set of disjoint TD trails; $\boldsymbol{\tau}$: new TD trail (possibly $\boldsymbol{\tau} \in \mathfrak{T}$)
**Output:** $\mathfrak{T}'$: updated set of disjoint TD trails that contains all new (non-truncated) trails from $\boldsymbol{\tau}$ (possibly $\mathfrak{T} = \mathfrak{T}'$)
procedure tdiff_absorb_new_trail($\mathfrak{T}, \boldsymbol{\tau}$)
// initialize a set $\mathbf{t}$ of TD trails with the input trail $\boldsymbol{\tau}$
$\mathbf{t} \leftarrow \emptyset$; add $\boldsymbol{\tau}$ to $\mathbf{t}$
for all $\mathbf{T} \in \mathfrak{T}$ do
    if $\mathbf{t} = \emptyset$ then
        // all trails in $\mathbf{t}$ have been fully absorbed; return
        return $\mathfrak{T}$
    end if
    // absorb $\mathbf{t}$ into $\mathbf{T}$ and store the remainder in $\mathbf{t}'$
    $\mathbf{t}' \leftarrow \emptyset$
    for all $\boldsymbol{\tau} \in \mathbf{t}$ do
        // if $\boldsymbol{\tau}$ contains trails not already in $\mathbf{T}$, then split $\boldsymbol{\tau}$ into TD trail subsets to exclude duplicates using
        if $(\mathbf{T} \subset \boldsymbol{\tau}) \vee (\mathbf{T}, \boldsymbol{\tau} : \mathsf{PO})$ then
            $\mathbf{t}_{\mathrm{temp}} \leftarrow$ tdiff_madd_trails_make_disjoint($\mathbf{T}, \boldsymbol{\tau}$)
            add $\mathbf{t}_{\mathrm{temp}}$ to $\mathbf{t}'$
        end if
        // if $\boldsymbol{\tau}, \mathbf{T}$: disjoint, then all trails in $\boldsymbol{\tau}$ are new, so add it
        if $(\boldsymbol{\tau}, \mathbf{T})$: disjoint then
            add $\boldsymbol{\tau}$ to $\mathbf{t}'$
        end if
        // if $\boldsymbol{\tau}$ is a subset of $\mathbf{T} \implies$ it contains no new trails, so do nothing
        if $(\boldsymbol{\tau} \subset \mathbf{T})$ then
            continue
        end if
    end for
    // overwrite $\mathbf{t}$ with the part of it that was not absorbed i.e. $\mathbf{t}'$
    $\mathbf{t} \leftarrow \mathbf{t}'$
end for
// $\mathbf{t}$ contains all trails not absorbed in $\mathfrak{T}$ – add them to $\mathfrak{T}$ and return
$\mathfrak{T}' \leftarrow \mathfrak{T} \cup \mathbf{t}$
return $\mathfrak{T}'$
```

6.11.3 Clustering Around Sub-optimal Trails

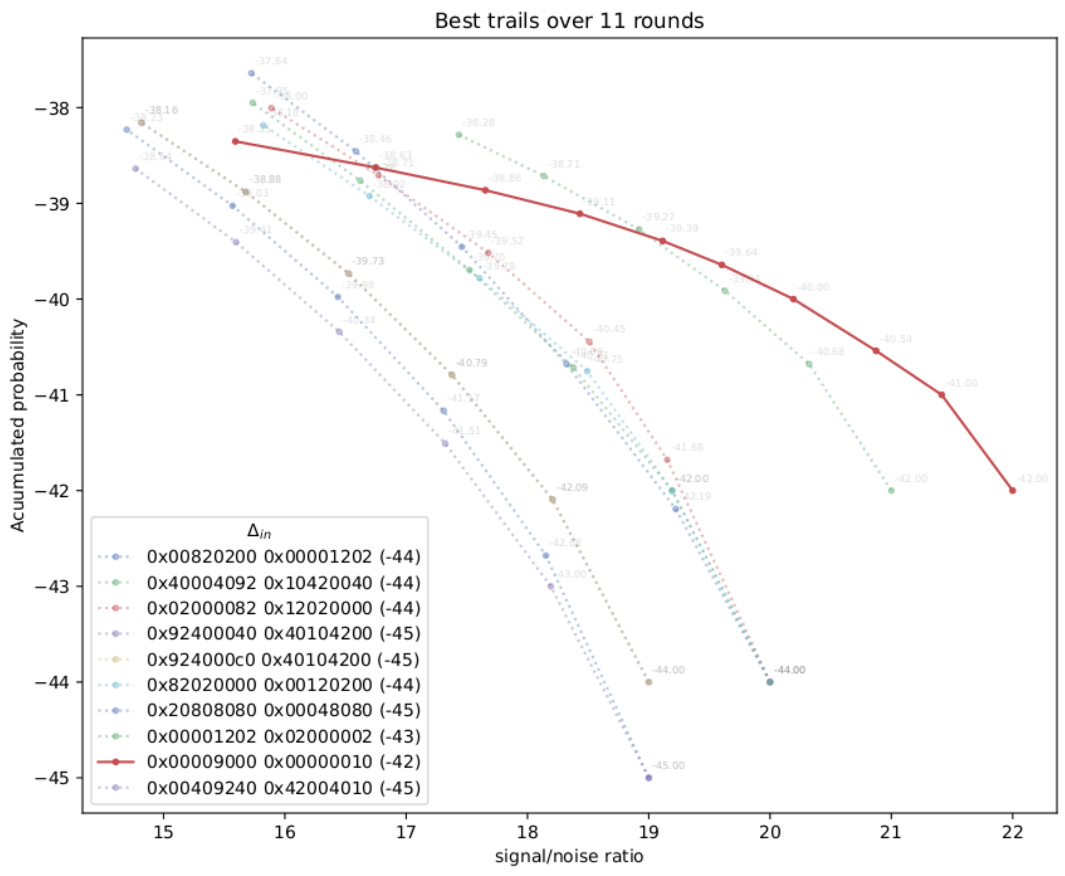

FIGURE 6.2: Trail probability and signal to noise ratio of clustering around 10 round seed trails. The solid-lines are those of the best seed trails, while the dotted ones are sub-optimal. The number near each vertex is the weight of the cluster up to this point. Notice that some trails, after clustering around them, result in better probabilities than the clustering around the best trail (for example, -42 near the start of the dotted green line shows that there are two trails of weight -43 at this point). The legend indicates the input differences to the cluster, in hexadecimal, with the probability of the seed trail in parenthesis.

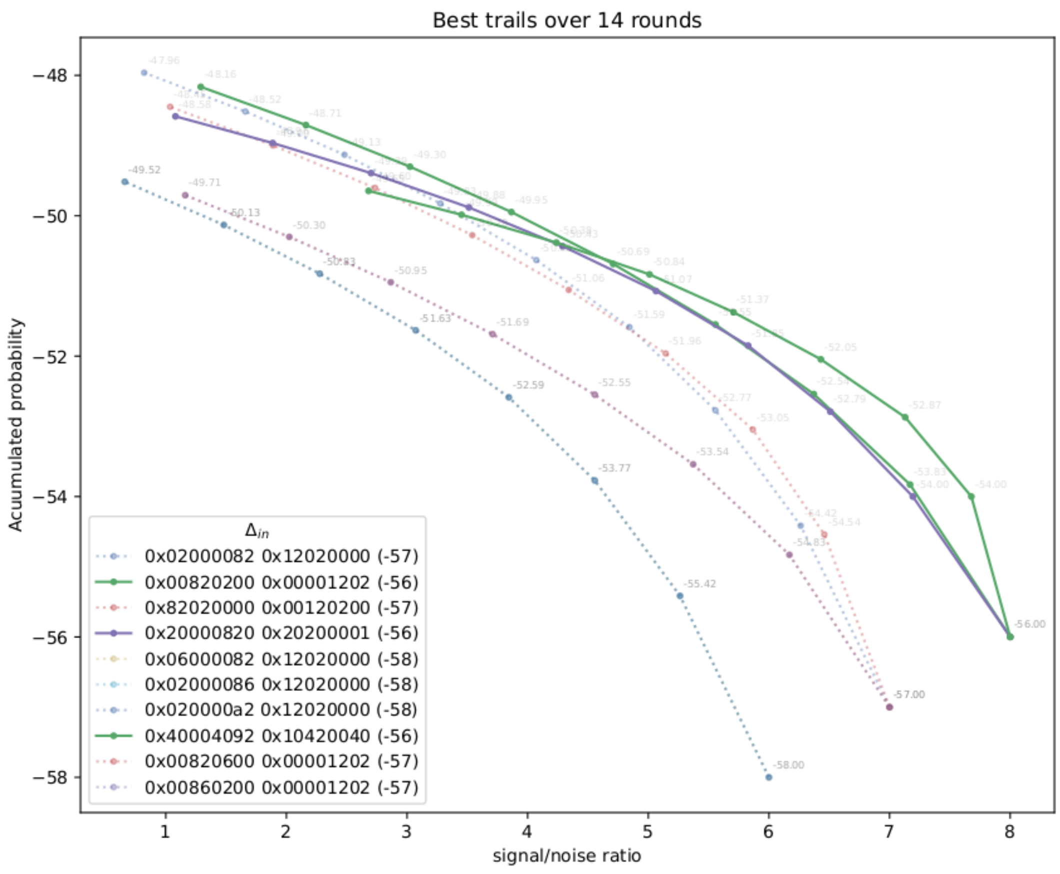

FIGURE 6.3: Trail probability and signal to noise ratio of clustering around 10 round seed trails. The solid-lines are those of the best seed trails, while the dotted ones are sub-optimal. The legend indicates the input differences to the cluster, in hexadecimal, with the probability of the seed trail in parenthesis ($\log_2$).

6.11.4 Examples of 6 Round Truncated Trails for Speck32

In Table 6.4 is shown an example of a strongly truncated trail (i.e. relatively many truncated bits) on 6 rounds SPECK32 produced from an initial suboptimal trail. Capital letter labels represent the negated value of the corresponding small letter label e.g. $\texttt{A} = 1 \oplus \texttt{a}$.

R	Star representation	Label representation	Pr
0	`0010001000000100`	`0010001000000100`	(-4.00)
	`0000101000000100`	`0000101000000100`	
	`0010100000000000`	`0010100000000000`	
1	`0000000001010000`	`0000000001010000`	(-6.00)
	`0000000000010000`	`0000000000010000`	
	`0000000001000000`	`0000000001000000`	
2	`1000000000000000`	`1000000000000000`	(-6.00)
	`0000000000000000`	`0000000000000000`	
	`1000000000000000`	`1000000000000000`	
3	`0000000100000000`	`0000000100000000`	(-6.42)
	`1000000000000000`	`1000000000000000`	
	`100000*100000000`	`100000a100000000`	
4	`0000000100000*10`	`0000000100000a10`	(-8.83)
	`100000*100000010`	`100000a100000010`	
	`1000000000000000`	`1000000000000000`	
5	`0000000100000000`	`0000000100000000`	(-11.16)
	`1000*10000001010`	`1000a10000001010`	
	`100001*1000*1*10`	`100001d1000c1b10`	

TABLE 6.3: Example of a 6 round truncated trail for SPECK32 with $\Pr = 2^{-11.16}$ obtained from the optimal non-truncated trail with $\Pr = 2^{-13}$. The labelled representation (right) indicates the dependency between the stars in the star representation (left). Numbers in brackets are the cumulative probabilities up to the corresponding round ($\log_2$ scale). The trail is expressed in terms of a sequence of three 16-bit values representing the two inputs and one output of the modular addition operation at each round.

R	Star representation	Label representation	Pr
0	`0000000000000000`	`0000000000000000`	(0.00)
	`1000000000000000`	`1000000000000000`	
	`1000000000000000`	`1000000000000000`	
1	`0000000100000000`	`0000000100000000`	(-1.42)
	`1000000000000010`	`1000000000000010`	
	`1000000100000*10`	`1000000100000a10`	
2	`0000*10100000010`	`0000a10100000010`	(-6.09)
	`1000000100001*00`	`1000000100001a00`	
	`1000010000*1*110`	`1000010000c1b110`	
3	`0*1*110100001000`	`0c1b110100001000`	(-14.09)
	`1000000000***100`	`1000000000CAb100`	
	`*110110100010100`	`d110110100010100`	
4	`0010100*11011010`	`0010100d11011010`	(-22.51)
	`*1101101***00110`	`d1101101CAb00110`	
	`00111011001***00`	`00111011001gfe00`	
5	`01***00001110110`	`01gfe00001110110`	(-33.34)
	`100011***01****1`	`100011cab01GFed1`	
	`**11010110111001`	`ih11010110111001`	

TABLE 6.4: Example of a 6 round truncated trail for SPECK32. The labelled representation (right) indicates the dependency between the stars in the star representation (left). Numbers in brackets are the cumulative probabilities up to the corresponding round. Capital letter labels represent the negated value of the corresponding small letter label e.g. `A` $= 1 \oplus$ `a`. The trail is expressed in terms of a sequence of three 16-bit values representing the two inputs and one output of the modular addition operation at each round.

6.11.5 Best 15 Round Distinguisher for Speck64 using Simple Truncation Rules

For completeness we list the output differences that correspond to the 22 non-truncated trails with $\Pr \geq 2^{-64}$ that compose the 15 round truncated set distinguisher for SPECK64 with probability $2^{-59.05}$ (Table 6.1) in Table 6.5.

i	$\Delta_{\mathrm{OUT}} = (\gamma^{15} \parallel \beta^{15})$	$\log_2 \Pr_i$	$\sum_i \log_2 \Pr_i$
0	0A080808 02084008	−62	−62.00
1	0A088808 02084008	−63	−61.42
2	0E080808 02084008	−63	−61.00
3	0A180808 02084008	−63	−60.69
4	1A080808 02084008	−63	−60.42
5	0A081808 02084008	−63	−60.19
6	0A080818 02084008	−63	−60.00
7	1A088808 02084008	−64	−59.91
8	0A089808 02084008	−64	−59.83
9	0A181808 02084008	−64	−59.75
10	0A081818 02084008	−64	−59.68
11	1E080808 02084008	−64	−59.61
12	0A188808 02084008	−64	−59.54
13	1A081808 02084008	−64	−59.48
14	0E081808 02084008	−64	−59.42
15	0A180818 02084008	−64	−59.36
16	1A080818 02084008	−64	−59.30
17	0A088818 02084008	−64	−59.25
18	1A180808 02084008	−64	−59.19
19	0E180808 02084008	−64	−59.14
20	0E080818 02084008	−64	−59.09
21	0E088808 02084008	−64	−59.05

TABLE 6.5: Output differences (in hexadecimal) corresponding to the 22 non-truncated trails for 15 rounds of SPECK64 with input difference (to the first round ADD) $\alpha^0 =$ 92400040 $\beta^0 =$ 10420040. These trails are a subset of the 135 trails in the set T_{ntr} in the third to last line of Table 6.1.

6.11.6 Optimal 15 rounds Trail for best Speck64 Distinguisher

i	Δx_i	Δy_i	Pr_i	$\sum_i \text{Pr}_i$
0	40004092	10420040		
1	82020000	00120200	-5	-5
2	00900000	00001000	-4	-9
3	00008000	00000000	-2	-11
4	00000080	00000080	-1	-12
5	80000080	80000480	-1	-13
6	00800480	00802084	-3	-16
7	80806080	848164a0	-6	-22
8	040f2400	20040104	-13	-35
9	20000820	20200001	-8	-43
10	00000009	01000000	-4	-47
11	08000000	00000000	-2	-49
12	00080000	00080000	-1	-50
13	00080800	00480800	-2	-52
14	00480008	02084008	-4	-56
15	0a080808	1a4a0848	-6	-62

TABLE 6.6: Optimal 15 round trail for SPECK64.

In Table 6.6 we report the optimal trail for SPECK64 reduced to 15 rounds we used in Section 6.9 to build a distinguisher with $S/N > 1$ and data complexity equal to $2^{-55.03}$.

Part V

Masked AES for microcontrollers

Chapter 7

MaskedAES

This chapter is based on the paper titled "Rivain-Prouff on Steroids: Faster and Stronger Masking of the AES", currently under review for publication.

This paper proposes a new method to efficiently compute a side-channel protected AES using the masking scheme described by Rivain and Prouff. The crucial part of this work is an improvement of the table-based multiplication in the field $\mathbb{F}_{2^8}$ used as a building block of the secure multiplication over shared values. Since the secure multiplication is performed multiple times to compute the inversion required by the non-linear layer of the scheme, any speed improvement has a strong impact on the overall performance of the masked scheme. Furthermore, we modify the sequence of secure multiplication to compute the inversion such that the number of table look-ups and the amount of randomness needed is minimized. To confirm the theoretical improvement, we study the scheme on an ARM Cortex-M4 microcontroller. We propose an assembly optimized implementation of the SubByte operation which result in a fully masked AES that is approximately two times faster than the results published at Eurocrypt 2017 by Goudarzi and Rivain. Since the primary goal of a masking scheme is not to be fast, but to be secure, we also performed a t-test on power traces measured on a real device in order to confirm that the method is sound.

On this chapter, my main contribution was the table-based multiplication, as well the optimization of the implementation using the new gadgets. I contributed with the practical aspect of the leakage assessment. The mathematical proofs of the gadgets were investigated mainly by my co-authors.

7.1 Introduction

Twenty years ago, the block cipher Rijndael was selected by NIST to become the encryption standard called AES. Since then, a lot of effort has been put into making efficient and secure implementations on various platforms. A common attack vector on concrete implementations is the exploitation of the physical behaviors of the device running the cryptographic algorithm. These so-called side-channel attacks (SCA) have been one of the focus of attention from cryptographers over the last two decades. While SCA is a broad term encompassing a large number of techniques used to extract information from a device, a well-studied sub-field is the statistical analysis of power consumption and electromagnetic emanations to find a correlation between the physical measurements and the secret values manipulated when performing cryptographic operations. A popular mitigation against such an attack is called *masking*, it aims to break the link between the internal intermediate values and the measurements by applying secret sharing techniques internally throughout the execution.

In masking, a secret value s is split into several statistically independent shares $(s_0, s_1, \dots, s_d)$ which hold the relation $s = s_0 \odot s_1 \odot \cdots \odot s_d$ for some operation $\odot$. We call d the *masking order*. The main challenge is finding an efficient algorithm that takes as input values split into shares and that computes correctly the result of the cryptographic transformations without ever recombining those shares (in the case of decryption, even the output still needs to be in shares form). Due to its popularity as a standard and widespread use, masking schemes for the AES have been developed and studied in the literature. Out of the four main operations of AES, three of them are trivial to mask, mainly due to linear nature. Indeed if $f(s_0) \odot f(s_1) \odot \cdots \odot f(s_d) = f(s_0 \odot s_1 \odot \cdots \odot s_d)$ holds for a cryptographic operation f, the non-masked version of the function can simply be applied independently to each share. Hence, the bulk of the work consists in finding an efficient way to compute the non-linear operation of AES (SubBytes) over the shares. Algebraically, this operation is an inversion in the field $\mathbb{F}_{2^8}$ (followed by an affine transformation which is trivial to mask), with the caveat that 0 is a valid input that is mapped to itself.

In this work, we aim to improve the practical performance of a technique proposed in 2010 by Rivain and Prouff ([RP10]) to compute the nonlinear substitution by computing $x \mapsto x^{254}$ in $\mathbb{F}_{2^8}$ that has been more recently studied by Goudarzi and Rivain ([GR17]). The efficiency of this technique is mainly driven by the efficiency of the multiplication in the field. Those multiplications are usually performed using the so-called Antilog and Log tables, which map elements to their order in the multiplicative group of the field and vice versa. To multiply two elements, one first maps them to their orders, adds the orders, and maps the result back to an element corresponding to their product. Unfortunately, since the work is done in the multiplicative group, multiplications by 0 need to be treated as special cases since 0 is not in the group. We remove this source of inefficiency by attributing a value corresponding to the element 0 in the Antilog table containing the orders. Using this trick, the field multiplication has no special case anymore and some instructions are saved. Furthermore, we use an alternate sequence of multiplications to compute the exponentiation to 254 that minimizes the number of look-ups in the Log and Antilog tables and that also requires less randomness to be secure in the probing model.

Related works. The main starting point is the paper of Rivain and Prouff [RP10]. In their work, they generalize the secure AND gate of Ishai, Sahai, and Wagner[ISW03] to perform a secure multiplication in $\mathbb{F}_{2^8}$. Using this multiplication, they construct an efficient algorithm to raise an element to the power 254. In [GR17], Goudarzi and Rivain study multiple techniques to mask AES from a practical point of view on the ARM architecture. Among others, they give results for implementations of AES using the method of Rivain and Prouff. Other masked implementations of AES on ARM can be found in [SS17; FPS17; JS17; Gro+19; AP20]. However, they use different masking techniques and/or are not possible to generalize to any order. Furthermore, there has been recent results indicating that a large amount of those existing masked AES implementations are actually flawed [Bec+22].

Our contribution. In this paper, we significantly improve the speed of the Rivain-Prouff masking scheme by proposing a more efficient table-based multiplication in $\mathbb{F}_{2^8}$ and by modifying the exponentiation in order to strongly synergize with our multiplication. In particular, we save some computation by completely removing the zero check for the inversion and we propose a new sequence of multiplications to compute x^{254} needing less randomness and requiring less table look-ups. We provide an assembly implementation and concrete speed measurements at order 1 as well as an experimental evaluation of the side-channel resistance of our implementation using a t-test on real traces. Our experimental results show that our code is almost twice as fast as the fastest version of Rivain-Prouff presented in [GR17] while exhibiting no leakage over 5000 traces.

7.2 Preliminaries

7.2.1 Advanced Encryption Standard

AES is essentially a substitution permutation network, where a 4×4 byte array (called *state*) is modified in a series of consecutive rounds. Specifically, each round is a composition of the following operations:

1. SubBytes: is the only non-linear transformation of the cipher. This transformation consists of a specific S-box applied to the state. Such S-box was designed to achieve non-linearity (minimizing the maximum input-output correlation amplitude and the maximum difference propagation probability) and algebraic complexity. The S-box is defined by the following function in $\mathbb{F}_{2^8}$: $\mathrm{Aff}_8(\mathrm{Inv}_8(\mathrm{a}))$, where Inv_8 is the inversion in the field and Aff_8 is an invertible affine transformation.
2. ShiftRows: is a byte transposition that cyclically shifts the rows of the state over different offsets (respectively 0, 1, 2 and 3 for the first, second, third and fourth row).
3. MixColumns: is a permutation of the state column by column. The columns of the state are multiplied by the polynomial $3x^3 + x^2 + x + 2 \mod x^4 + 1$.
4. AddRoundKey: is a XORing operation between the state and *round key*, namely each round uses its own round key. The array of round keys (called ExpandedKey[]) comes from a key schedule procedure taking as input the key of the cipher. The size of the ExpandedKey is equal to the block length multiplied by the number of rounds. In fact, the number of rounds depends on the key size (for a 128-bit key, 10 rounds are performed).

Hence, let us shortly outline the high-level structure of AES. An AES encryption consists in expanding the key followed by a first XOR between the the initial state and the cipher key. Afterward, the algorithm iterates the round procedure for $N_r - 1$ times, since the last round is composed slightly differently. Namely, for the last round the MixColumn is removed, as it adds no security to the cipher since it can be trivially inverted. See algorithm 24 the details.

Algorithm 24 AES - Encryption protocol

```
Rijndael(State,CipherKey)
    KeyExpansion(CipherKey, ExpandedKey)
    AddRoundKey(State, ExpandedKey[0])
    for $i = 1$ to $N_r - 1$ do
        SubBytes(State)
        ShiftRows(State)
        MixColumn(State)
        AddRoundKey(State, ExpandedKey[i])
    SubBytes(State)
    ShiftRows(State)
    AddRoundKey(State, ExpandedKey[$N_r$])
```

Algorithm 24 is only describing the encryption procedure since the decryption is straightforward. In fact, note that each of the above operations is invertible and we refer the reader to [DR02] for the decryption and the security proofs.

7.2.2 Provably-Secure Masking of the AES

Ishai, Sahai, and Wegner, in [ISW03], introduced and formalized a new model to ensure the security of any binary circuits against *probing attacks*. Their constructions exploit a $(d+1)$-*out-of*-$(d+1)$ secret sharing scheme, where observing any subset of d shares reveals nothing about the original value. Such technique has been called binary masking and each intermediate variable x is mapped into a set of $d+1$ shares such that $x = x_0 \oplus x_1 \oplus \cdots \oplus x_d$, where $x_1, x_2, \ldots, x_d$ are generated at random while x_0 is computed accordingly [1]. Moreover, each share x_i is processed along the circuit separately and they proved security against any attacker probing $t = (d-1)/2$. Since [ISW03] has been proposed, the probing model has been widely studied and improved. In 2010, Rivain and Prouff in [RP10] showed how to securely mask AES, adapting the ISW construction to securely work in $\mathbb{F}_{2^8}$. For completeness, let us briefly mention the basics of the Rivain-Prouff masking of AES.

The AES block cipher is structured as multiple rounds transformation, consisting of a key addition, a linear and a non-linear layer. The masking of latter operation, i.e the masking of the S-box, is the trickiest part of the implementation. In particular, their main challenge was to provide a secure method to mask the power-to-254 function for any order d. The exponentiation to 254, as described in [DR02], consists of 4 multiplications over $\mathbb{F}_{2^8}$ and 7 squarings[2]. Hence, to securely compute such exponentiation, they had to implement a secure method for the squaring and the multiplication. While the former is straightforward (due to its linearity it is secure to square each share separately and independently), for the multiplication they proposed

[1]In order to not confuse the readers, we would like to point out that in our paper we adopted the Rivain-Prouff notation, namely: we refer to $d+1$ as the number of shares and to d as the masking order, as a consequence of that the indices for the shares will go from 0 to d. Please note that this is slightly different from the notation used in the Goudarzi-Rivain paper.

[2]Considering exponentiation to the power of 4 and 16, respectively, as 2 and 4 consequent squarings.

the following secure algorithm[3]:

Algorithm 25 SecMult - dth-order secure multiplication over $\mathbb{F}_{2^k}$

```
Input: shares $a_i$ satisfying $a = \bigoplus_{i=0}^{d} a_i$, shares $b_i$ satisfying $b = \bigoplus_{i=0}^{d} b_i$
Output: shares $c_i$ satisfying $c = \bigoplus_{i=0}^{d} c_i = ab$
1: for $i = 0$ to $d$ do
2:     for $j = i + 1$ to $d$ do
3:         $r_{i,j} \leftarrow \mathsf{rand}(k)$
4:         $r_{j,i} \leftarrow (r_{i,j} \oplus a_i b_j) \oplus a_j b_i$
5: for $i = 0$ to $d$ do
6:     $c_i \leftarrow a_i b_i$
7:     for $(j = 0$ to $d) \wedge (i \neq j)$ do
8:         $c_i \leftarrow c_i \oplus r_{i,j}$
```

Unfortunately as argued in [RP10], the secure computation of the squaring and the multiplication are not enough to guarantee the security for the whole exponentiation. In order to do that the masks of the inputs given to the SecMult needs to be mutually independent and this means they need to be refreshed in some point during the exponentiation. This operation is processed by the following secure refreshing algorithm:

Algorithm 26 RefreshMasks

```
Input: shares $x_i$ satisfying $x = \bigoplus_{i=0}^{d} x_i$
Output: shares $x_i$ satisfying $x = \bigoplus_{i=0}^{d} x_i$
1: for $i = 1$ to $d$ do
2:     $\mathsf{tmp} \leftarrow \mathsf{rand}(k)$
3:     $x_0 \leftarrow x_0 \oplus \mathsf{tmp}$
4:     $x_i \leftarrow x_i \oplus \mathsf{tmp}$
```

Thus, finally, let's recall their secure algorithm for the power-to-254 function in Algorithm 27. Note that from the following secure exponentiation one can easily extend the security notion to the whole S-box. This is due to the fact the affine transformation called after the exponentiation is linear and it can be computed (securely) dealing with each share separately, namely:

$$Af(x_0) \oplus Af(x_1) \oplus \cdots \oplus Af(x_d) = \begin{cases} Af(x) & \text{if } d \text{ is even,} \\ Af(x) \oplus \mathsf{0x63} & \text{if } d \text{ is odd.} \end{cases}$$

Concerning the remaining AES operations, their implementation is quite straightforward and we refer the readers to look at the original Rivain-Prouff paper. While Algorithm 27 is secure at order 1, it should be noted that a flaw has been found and described in [Cor+14] for higher orders. Since we propose a new exponentiation together with a security proof, Algorithm 27 is only recalled to ease the exposition and we can ignore this issue.

[3] We refer the readers to [RP10] for the security proof.

Algorithm 27 SecExp254 - dth-order secure exponentiation to the 254 over $\mathbb{F}_{2^k}$

Input: shares $x_0, x_1, \ldots, x_d$ satisfying $x = \bigoplus_{i=0}^{d} x_i$
Output: shares $y_0, y_1, \ldots, y_d$ satisfying $y = \bigoplus_{i=0}^{d} y_i = x^{254}$

```
1: for $i = 0$ to $d$ do $z_i \leftarrow x_i^2$ // $\bigoplus_i z_i = x^2$
2: $(x_0, x_1, \ldots, x_d) \leftarrow \mathsf{RefreshMask}(x_0, x_1, \ldots, x_d)$
3: $(y_0, y_1, \ldots, y_d) \leftarrow \mathsf{SecMult}((z_0, z_1, \ldots, z_d), (x_0, x_1, \ldots, x_d))$ // $\bigoplus_i y_i = x^3$
4: for $i = 0$ to $d$ do $w_i \leftarrow y_i^4$ // $\bigoplus_i w_i = x^{12}$
5: $(w_0, w_1, \ldots, w_d) \leftarrow \mathsf{RefreshMask}(w_0, w_1, \ldots, w_d)$
6: $(y_0, y_1, \ldots, y_d) \leftarrow \mathsf{SecMult}((y_0, y_1, \ldots, y_d), (w_0, w_1, \ldots, w_d))$ // $\bigoplus_i y_i = x^{15}$
7: for $i = 0$ to $d$ do $y_i \leftarrow y_i^{16}$ // $\bigoplus_i y_i = x^{240}$
8: $(y_0, y_1, \ldots, y_d) \leftarrow \mathsf{SecMult}((y_0, y_1, \ldots, y_d), (w_0, w_1, \ldots, w_d))$ // $\bigoplus_i y_i = x^{252}$
9: $(y_0, y_1, \ldots, y_d) \leftarrow \mathsf{SecMult}((y_0, y_1, \ldots, y_d), (z_0, z_1, \ldots, z_d))$ // $\bigoplus_i y_i = x^{254}$
```

7.3 Low-Level Field Arithmetic

A masked implementation of the AES based on the RP technique performs the inversion in $\mathbb{F}_{2^8}$, which is part of the SubBytes transformation, through exponentiation by the exponent 254 according to Fermat's theorem. Consequently, the inversion boils down to a sequence of multiplications and squarings in $\mathbb{F}_{2^8}$. In essence, a multiplication of two elements of $\mathbb{F}_{2^8}$ consists of a multiplication of binary polynomials of a degree of up to 7, yielding a product-polynomial with a maximum degree of 14, followed by a reduction modulo the irreducible polynomial $p(x) = x^8 + x^4 + x^3 + x + 1$. Modern x64 processors provide the `GF2P8MULB` instruction for multiplication in $\mathbb{F}_{2^8}$, but such an instruction is lacking on our target platform, the ARM Cortex-M3, and also on all other 8, 16, and 32-bit microcontrollers. Therefore, implementers have to resort to either the so-called shift-and-XOR method, which is relatively slow, or performs the multiplication with the help of look-up tables.

7.3.1 Table-Based Multiplication

In its most simple form, a table-based multiplication of elements of $\mathbb{F}_{2^8}$ uses two look-up tables, namely a so-called *Log table* that contains the order of the 255 non-0 elements of the field (based on the generator $g(x) = x + 1$) and an *AntiLog table* containing the elements corresponding to the orders in the range of $[0, 254]$. Each of the tables consists of 255 entries, but for efficiency reasons a Log table with 256 entries is used so that the integer representation of a field element can be directly used as an index for the look-up into the table. Note that only the elements of the multiplicative group of $\mathbb{F}_{2^8}$ (i.e. the non-0 elements of $\mathbb{F}_{2^8}$) actually have an order. However, by assigning 255 to the Log-table entry with index 0 and by assigning 0 to the AntiLog-table entry with index 255, it is guaranteed that `AntiLog[Log[a]] = a` holds for *any* $a(x) \in \mathbb{F}_{2^8}$, including the special case $a(x) = 0$. When implemented in this way, both the Log table and the AntiLog table have a size of 256 bytes, which amounts to 512 bytes altogether. The computation of the product $c(x) = a(x)b(x)$ consists of three basic steps. At first, the Log table is used to obtain $u = \mathrm{ord}(a(x))$ and $v = \mathrm{ord}(b(x))$. Then, the modular sum $s = u + v \bmod 255$ is computed by means of a conventional 8-bit addition followed by a conditional subtraction of 255. Finally, the field-element $c(x)$ corresponding to the order s is determined with the help of the AntiLog table. Consequently, the multiplication requires three table look-ups and a modular addition [DR02; Gla07].

It is possible to speed up the multiplication by replacing the modular addition with an ordinary addition, but this requires a larger AntiLog table [Gla07]. Concretely, when the AntiLog table is duplicated so that `AntiLog[255+i] = AntiLog[i]` for $i \in [0, 254]$, then the conditional subtraction of 255 from $u + v$ is not necessary anymore. Thanks to this simple optimization, the cost of the table-based multiplication is reduced to three table look-ups and an ordinary addition. The size of the AntiLog table doubles, and both tables amount to 768 bytes altogether. Note that, since the two tables are static, they can be kept in non-volatile memory (i.e. ROM or flash) and do not necessarily occupy precious space in RAM. However, there is a further implementation detail that requires special attention, namely the treatment of the case $a(t) = 0$ or $b(t) = 0$, which we ignored until now. Unfortunately, the table-based multiplication as described above does not yield the correct result when one of the operands is 0. A simple mathematical explanation for this observation is the fact that 0 is not an element of the multiplicative group of the finite field $\mathbb{F}_{2^8}$ and, therefore, it does not have a multiplicative order. Software implementations usually treat the occurrence of 0-operands as a special case and explicitly set the result to 0 when one of the two operands is 0. Hence, a table-based multiplication involves besides the three steps already mentioned above a fourth step in which the operands are checked for 0 and the result is corrected if needed. A pseudo-code description of the table-based multiplication in $\mathbb{F}_{2^8}$ can be found at the end of Subsection 7.2 of [Gla07].

Listing 7.1: ARM Assembler macro for table-based multiplication in $\mathbb{F}_{2^8}$ (see [GR17]).

```
.macro F2P8MUL res:req, opA:req, opB:req, tmp:req, ptLog:reg, ptALog:req
// step 1: tmp <- Log[opA], res <- Log[opB]
ldrb    \tmp, [\ptLog, \opA]
ldrb    \res, [\ptLog, \opB]
// step 2: tmp <- tmp + sum
add     \tmp, \tmp, \sum
// step 3: res <- ALog[tmp]
ldrb    \res, [\ptALog, \tmp]
// step 4: treat (opA = 0) or (opB = 0) case
rsb     \tmp, \opA, #0            // tmp <- 0 - opA
and     \tmp, \opB, \tmp, asr #32 // tmp <- opB & (tmp >> 32)
rsb     \tmp, \tmp, #0            // tmp <- 0 - tmp
and     \res, \res, \tmp, asr #32 // res <- res & (tmp >> 32)
.endm
```

Listing 7.1 contains an optimized implementation of the table-based multiplication in ARMv7M Assembly language (GNU syntax) that can be executed on e.g. Cortex-M3 and Cortex-M4 microcontrollers. This implementation is based on the assembly code provided in Appendix A (concretely in Figure 24) of [GR17], which is an extended version of the EUROCRYPT 2017 paper of Goudarzi and Rivain [GR17]. It follows the four basic steps described above but differs slightly from the original Assembler code of Goudarzi and Rivain. For example, our version does not contain the reduction of the sum of the orders modulo 255 since we take advantage of the "duplicated" AntiLog table, which saves an `add` as well as an `and` instruction. Furthermore, our implementation does not contain the `ldr` ("load register") instruction at the start of Goudarzi and Rivain's Assembler macro (used to initialize the register `ptLog` with the pointer to the Log table) since we assume that `ptLog` already contains the Log-table address whenever the macro is executed. The last four instructions do a 0-testing of the operands and set the result in register `res` to 0 when (at least) one of the operands is 0. These four instructions are exactly the same as in Subsection 3.2 of [GR17; GR17] and in Figure 24 of [GR17]. The result of the `rsb` ("reverse subtract") instruction at line 10 is either 0 (when register `opA` was 0) or the two's complement of `opA`, in which case its most-significant bit is 1. Note that the second operand of the subsequent `and` instruction (i.e. register `tmp`) is *arithmetically* shifted to the right before the actual AND operation. Hence, the second operand is either 0 or the "all-1" word $2^{32}-1$, i.e. `0xffffffff` in hex notation, and the content of register `tmp` is either 0 or `opB`. After execution of the other two `rsb` and `and` instructions in line 12 and 13, respectively, the content of `res` is either 0 or the multiplication result from line 8.

The `ldrb` ("load register with byte") instruction has a latency of two clock cycles[4] on our target device, which is a VL Discovery development board from STMicroelectronics housing a STM32F100RBT6B Cortex-M3 microcontroller that is clocked with a nominal frequency of 24 MHz [STM16]. All other instructions in Listing 7.1 have a latency of one cycle. Hence, the F2P8MUL macro has an overall execution time of 11 cycles, of which four are spent for the special treatment of 0-operands. These four clock cycles constitute a whopping 36.4% of the execution time and have a massive impact on the performance of the RP masking scheme. Namely, as we will see in Section 7.5 using first-order masking as case study, the inversion of the SubBytes transformation represents about 90% of the execution time of an AES round. The execution time of the inversion depends heavily on that of the masked multiplication (i.e. SecMul operation), which, in turn, is dominated by ordinary multiplication in $\mathbb{F}_{2^8}$ (i.e. the `F2P8MUL` macro). Therefore, getting rid of the special treatment of 0-operands in Listing 7.1 would lead to a significant reduction of the execution time of an RP-masked AES. However, this seems to be a non-trivial problem since neither Goudarzi and Rivain nor Rivain and Prouff found a solution.

Eliminating the Special Treatment of 0-Operands. In the following, we show that it is possible to completely avoid the special treatment of 0-elements, thereby improving the efficiency of RP masking by a massive extent, at the cost of an increased size of both the Log table and the AntiLog table. Although our idea is surprisingly simple, did not see it published anywhere, neither in the landmark paper of Rivain and Prouff [RP10], nor in any subsequent paper on optimized software implementation of the RP masking scheme such as [GR17]. Recall that the very first entry of our Log table (i.e. the entry with the index 0) is 255. Furthermore, as also explained before, the entry with index 255 of the original (un-duplicated) AntiLog table is 0 to ensure `AntiLog[Log[0]] = 0`. Our idea is based on the fact that *the sum of the order of two non-0 elements of* $\mathbb{F}_{2^8}$ *is strictly less than 509.* Namely, when we modify the Log table by setting the very first entry to 509 (which could be interpreted as assigning the field-element 0 the order 509, although this makes no sense from a mathematical point of view), the special treatment of 0-operands can be simply avoided if the AntiLog table is modified accordingly. This means we have to extend the AntiLog table from 512 to 1024 entries, whereby the upper half of this extended table (i.e. all entries with index 509 to index 1023) contain 0. Note that these modifications do not impact the multiplication of non-0 elements since it still works in exactly the same way as before and yield the correct result. However, when one of the two operands is 0, the sum of the orders is at least 509, which implies an entry in the upper part of the AntiLog table is accessed and the result of the multiplication is 0.

These simple modifications of the Log and AntiLog table make it possible to remove the last four instructions from Listing 7.1, which reduces the overall execution time

[4]On certain Cortex-M models, neighboring load and store instructions can pipeline their address and data phases, which enables these instructions to complete in a single execution cycle.

of the F2P8MUL macro by 36.4%. However, this speed-up comes at the expense of larger look-up tables. The Log table still consists of 256 entries, but needs to be able to accommodate the 9-bit integer 509, which means it has to be of 16-bit type hword instead of 8-bit type byte. Furthermore, the AntiLog table requires 1024 entries instead of 512, whereby all entries in the upper half (i.e. index 509 and above) are 0. Hence, the two tables become twice as large, i.e. the size of both tables amounts to 1536 bytes altogether, which is still relatively small compared to the non-volatile memory consumption of other masked AES implementations (as we will see in Section 7.5).

[5]

Reducing the Number of Table Look-Ups. The exponentiation-based secure inversion from the Rivain-Prouff paper [RP10], provided in Algorithm 27, consists of four masked multiplications (so-called SecMult gadgets), three other masked arithmetic operations in $\mathbb{F}_{2^8}$ (i.e. squaring, fourth-power, 16th-power[6]), and two mask refreshings. However, the three other arithmetic operations are all linear in $\mathbb{F}_{2^8}$ and can, therefore, be performed separately on each of the $d+1$ shares. In addition, the three operations have in common that each of them can be implemented with a 256-byte look-up table. For example, the squaring is an ordinary table look-up, which, given any $a \in \mathbb{F}_{2^8}$, simply returns the field-element $b = a^2$ as result, i.e. it is not necessary to obtain the order of a first like in the multiplication. When we ignore the mask refreshings, the first two steps of the inversion in Algorithm 27 are a squaring followed by a multiplication[7], and this operation sequence (i.e. *a multiplication preceded by a linear operation* that requires just an ordinary table look-up) appears two further times, namely in lines 4 and 6, and in lines 7 and 8. Based on this observation, it is possible to reduce the total number of table look-ups during an exponentiation by modifying the squaring table (resp. fourth-power/16th-power table) to contain *the order of the square* $\mathrm{ord}(b)$ *instead of the actual square* b. This modification does not increase the size of the table, but allows one to accelerate the exponentiation by using the order read from the table as operand in a subsequent multiplication, which saves a look-up into the Log table. In the ideal case, i.e. when both operands are given as orders, the multiplication only consists of an addition and a single table look-up.

This optimization to reduce the number of table look-up is not particularly effective when applied to the original masked exponentiation given in Algorithm 26 because of the mask refreshings between the squaring and multiplication at line 2 and line 5 (which we ignored in the above description of our idea). However, in Section 7.4, we will introduce a modified variant of the exponentiation that is much better suited for our optimization and, therefore, faster than the original exponentiation of Rivain and Prouff.

[5]The modified Log and AntiLog table as well as a C implementation of the optimized multiplication (without special treatment of 0-operands) can be found in the full paper and online, after the paper is over the anonymous evaluation phase.

[6]As mentioned in Section 7.2, the fourth-power function $x \mapsto x^4$ and 16th-power function $x \mapsto x^{16}$ can be implemented via squarings, in which case seven squarings have to be performed in an inversion.

[7]The be precise, these two steps are a *masked* squaring and *masked* multiplication. However, for the explanation of our optimization, it does not matter whether these operations are masked or not.

7.3.2 Basic First-Order Gadgets

As explained in Section 7.2, the RP masking scheme is generic in the sense that it can be used to achieve arbitrary masking orders, thereby enabling different trade-offs between execution time and security, i.e. resistance to DPA attacks. However, in the rest of this section, we focus on first-order masking, i.e. we assume $d + 1 = 2$ shares, to simplify the description of how we implemented the basic gadgets for masked operations in $\mathbb{F}_{2^8}$. We emphasize that all optimizations described in this section translate over to higher orders in a natural fashion.

In the case of first-order masking (i.e. $d = 1$), all sensitive variables are to be split up into $d+1 = 2$ shares. Consequently, the two operands $a, b \in \mathbb{F}_{2^8}$ that are input to the secure (i.e. masked) multiplication given in Algorithm 25 have the form $a = a_0 \oplus a_1$ and $b = b_0 \oplus b_1$, respectively. For $d = 1$, the computation of the two shares c_0 and c_1 of the product $c = ab = c_0 \oplus c_1$ involves four conventional multiplications in $\mathbb{F}_{2^8}$ as follows.

$$\begin{aligned} &r_{0,1} = \mathsf{rand}(8bit), \;\; r_{1,0} = (r_{0,1} \oplus a_0 b_1) \oplus a_1 b_0 \\ &c_0 = a_0 b_0 \oplus r_{0,1}, \;\; c_1 = a_1 b_1 \oplus r_{1,0} \end{aligned} \tag{7.1}$$

As explained in the last subsection, it is possible to have a Log table and AntiLog table so that `AntiLog[Log[a]] = a` holds for any $a \in \mathbb{F}_{2^8}$ (including 0) and not just the non-0 elements. Consequently, it is possible to define a bijective mapping between the elements of $\mathbb{F}_{2^8}$ and their orders, even though, strictly speaking, the element 0 does not have an order in a mathematical sense. This means it is possible to represent any element by its order and vice versa, and we can easily convert between these representations, which we call the *normal domain* and the *order domain*, with the help of look-up tables.

Listing 7.2: ARM Assembler macro of a two-share SecMult gadget (the shares a_0, a_1 and b_0, b_1 of the operands a and b are in the order domain, whereas the shares c_0, c_1 of the product $c = ab$ are in the normal domain).

```
.macro SECMULT c0:req,c1:req, a0:req,a1:req, b0:req,b1:req, rn:req, ptALog:req
// products in order domain
add     \c0, \a0, \b0
add     \c1, \a1, \b1
add     \a0, \a0, \b1
add     \a1, \a1, \b0
// products in normal domain
ldrb    \c0, [ptALog, \c0]
ldrb    \c1, [ptALog, \c1]
ldrb    \a0, [ptALog, \a0]
ldrb    \a1, [ptALog, \a1]
// compute masked result c0,c1
eor     \c0, \c0, \rn
eor     \rn, \a0, \rn
eor     \rn, \a1, \rn
eor     \c1, \c1, \rn
.endm
```

Listing 7.2 shows an ARM Assembly macro that implements a SecMult gadget for the first-order case, i.e. $d = 1$, based on Equation (7.1). This implementation incorporates the two optimizations we presented in the last subsection, i.e. it does not treat 0-operands in a special way and it assumes the two operands (or, more precisely, the shares of the operands) to be in the order domain, whereas the shares of the result are in the normal domain so as to reduce the overall number of table look-ups in a masked inversion. Since the macro is based on Equation (7.1), it basically performs four multiplications in $\mathbb{F}_{2^8}$ and four XOR operations. However, since the operands are given in the order domain, the multiplications to produce a_0b_0, a_1b_0, a_0b_1, and a_1b_1 are simply additions of the orders (i.e. the `add` instructions from line 3 to 6). The subsequent `ldrb` instructions convert the four products from the order domain to the normal domain through look-ups into the AntiLog table (similar to Listing 7.1, the register `ptALog` contains the start address of the table). Finally, the four `eor` instructions compute the two shares c_0, c_1 of the result c in the normal domain, whereby register `rn` holds a random byte.

The 12 instructions of the `SECMULT` macro have an execution time of 16 clock cycles when assuming that each `ldrb` has a latency of two cycles. On the other hand, a first-order SecMult gadget implemented as described in [GR17; GR17] would carry out the multiplication in $\mathbb{F}_{2^8}$ with help of the `F2P8MUL` macro shown in Listing 7.1, which has an execution time of 11 cycles. Four executions of `F2P8MUL` along with four `eor` instructions would result in an overall execution time of 48 cycles. Hence, the two optimizations we proposed in the last subsection, which eliminate the special treatment of 0-operands and reduce the overall number of table look-ups, make the first-order SecMult gadget three times faster compared to the best implementation in the literature.

In addition to SecMult, a masked inversion in $\mathbb{F}_{2^8}$ requires a few further gadgets, all of which perform linear operations and are, therefore, relatively easy to implement. The most important ones[8] are those for the masked squaring and 16th-power function; we call these gadgets SecSquare and SecPow16, respectively. Both have in common that the two shares of the input are in the normal domain, while the shares of the output are in the order domain such that they can be directly fed into a subsequent SecMult gadget. The underlying squaring table has to contain the order of the squares instead of the actual squares, and this also applies to the 16th-power table. Since the operations are linear, we can simply perform them on each share separately, i.e. the corresponding Assembly code consists of just two `load` instructions. A further gadget needed by the exponentiation is RefeshMasks, which injects fresh randomness to the masked representation of a sensitive variable. Finally, we also implemented a few auxiliary gadgets for such functions like the conversion of operands between the normal domain and the order domain.

7.4 New Exponentiation-Based Inversion

The exponentiation $x \mapsto x^{254}$ is the main operation of our masking scheme. It is crucial to find the most efficient way to compute it using the $\mathbb{F}_{2^8}$ multiplication technique described in the previous section, that is to say, minimizing the number table look-ups by minimizing the number of conversions between the normal domain and the order domain. There are four main operations ("gadgets") that are used in a secure exponentiation: SecMult, SecSquare, SecPow16, and RefreshMasks. The optimal representation for those operations is as follows:

1. RefreshMasks: As mentioned before, this operations injects fresh randomness into the shares. Since there is no trivial way to do it from the order domain,

[8]An efficient implementation of the original exponentiation in Algorithm 27 also needs a gadget for the 4th-power function, but this is not the case for the new exponentiation we present in the next section.

the inputs and the outputs should be in the normal domain.

2. SecMult: Since the first operations in the SecMult algorithm are ordinary multiplications in $\mathbb{F}_{2^8}$, the order domain is preferred for the inputs. Conversely, the last step is somewhat similar to a mask refreshing and, thus, the outputs are in the normal domain.

3. SecSquare and SecPow16: These are the most flexible operations. Since the tables are pre-computed and the look-ups are performed share by share, there is no efficiency difference between the two representations and and implementer is free to chose them for inputs and outputs, but of course the look-up tables have to be generated accordingly.

Algorithm 28 new_SecExp254 computation

```
Input: shares $x_0, x_1, \ldots, x_d$ satisfying $x = \bigoplus_{i=0}^{d} x_i$
Output: shares $e_0, e_1, \ldots, e_d$ satisfying $e = \bigoplus_{i=0}^{d} e_i = x^{254}$
1: $(z_i)_{0\le i\le d} \leftarrow \mathsf{SecSquare}((x_i)_{0\le i\le d})$ // $\bigoplus_i z_i = x^2$
2: $(z_i)_{0\le i\le d} \leftarrow \mathsf{RefreshMasks}((z_i)_{0\le i\le d})$
3: $(y_i)_{0\le i\le d} \leftarrow \mathsf{SecMult}((z_i)_{0\le i\le d}, (x_i)_{0\le i\le d})$ // $\bigoplus_i y_i = x^2 \cdot x = x^3$
4: $(z_i)_{0\le i\le d} \leftarrow \mathsf{SecSquare}((y_i)_{0\le i\le d})$ // $\bigoplus_i z_i = (x^3)^2 = x^6$
5: $(y_i)_{0\le i\le d} \leftarrow \mathsf{SecMult}((z_i)_{0\le i\le d}, (x_i)_{0\le i\le d})$ // $\bigoplus_i y_i = x^6 \cdot x = x^7$
6: $(z_i)_{0\le i\le d} \leftarrow \mathsf{SecSquare}((y_i)_{0\le i\le d})$ // $\bigoplus_i z_i = (x^7)^2 = x^{14}$
7: $(y_i)_{0\le i\le d} \leftarrow \mathsf{SecMult}((z_i)_{0\le i\le d}, (x_i)_{0\le i\le d})$ // $\bigoplus_i y_i = x^{14} \cdot x = x^{15}$
8: $(y_i)_{0\le i\le d} \leftarrow \mathsf{SecPow16}((y_i)_{0\le i\le d})$ // $\bigoplus_i y_i = (x^{15})^{16} = x^{240}$
9: $(e_i)_{0\le i\le d} \leftarrow \mathsf{SecMult}((y_i)_{0\le i\le d}, (z_i)_{0\le i\le d})$ // $\bigoplus_i e_i = x^{240} \cdot x^{14} = x^{254}$
```

In the original Rivain-Prouff exponentiation, the order of operations is such that, when using our gadgets, some extra conversions between representations would be needed. For example, at the end of Algorithm 27, the two SecMult operations are sequential, which means that a conversion from normal to order domain is required in between. To improve efficiency, we propose a new exponentiation algorithm in Algorithm 28. In this algorithm, we can see that the four SecMult operations are separated by SecSquare (resp. SecPow16) operations, which means that, when the squaring and 16th-power table are pre-computed to output the result in the order domain, no extra conversions are needed except after the RefreshMasks. Furthermore, one less RefreshMasks is carried out compared to the original exponentiation, which saves one byte of randomness and a couple of XORs.

The t-SNI property of our new exponentiation algorithm is out of the scope of this thesis, and the reader is invited to read the full paper for the proof [San+22].

7.5 Masked AES

In this section we provide a brief description of how we implemented the first-order masked round transformations of AES128 and report the execution time of each transformation on a Cortex-M3 microcontroller. Thereafter, we compare our results with that of some other masked AES implementations for the 32-bit ARM platform.

7.5.1 Round Transformations

We implemented all four round transformations, including the simple ones like ShiftRows and AddRoundKey, entirely in Assembly language to exclude the theoretical possibility that the compiler introduces leakage of sensitive data. Unlike some other implementations reported in the literature, we did not solely strive for high speed, but tried to achieve a reasonable trade-off between execution time and (binary) code size. Therefore, we refrained from certain optimization techniques like full loop-unrolling, which often produce only modest speed-ups at the expense of an unreasonably large increase of code size.

SubBytes is, from an implementers perspective, by far the most challenging round transformation because it has a significant impact on the overall execution time and is particularly leakage-sensitive due to its non-linear nature. The SubBytes transformation consists of 16 S-box operations, which, in turn, are composed of inversion in $\mathbb{F}_{2^8}$ along with an affine operation. Our masked inversion is based on the exponentiation technique we presented in Section 7.4 and uses (variants of) the optimized low-level gadgets introduced in Section 7.3. The proposed exponentiation has three of advantages over the original exponentiation method of Rivain and Prouff. First, it allows us to better exploit our optimization to reduce the overall number of table look-ups and is, therefore, faster. Second, it needs only one RefreshMasks gadget instead of two, thereby reducing the number of random bytes from six to five. Third, as explained in the last section, it achieves SNI, which is a stronger notion of probing security that is not met by the original (i.e. flawed) exponentiation algorithm from [RP10].

Even though our implementation of the masked exponentiation is, in essence, based on Algorithm 28, there are two aspects that deserve further explanations. First, for the sake of brevity, Algorithm 28 provides only a high-level description of our exponentiation using the four main gadgets (i.e. SecMult, SecSquare, SecPow16, and RefreshMasks), but omits gadgets for auxiliary operations like the conversion of operands between the normal domain and the order domain. For example, at the very beginning of the exponentiation, the shares x_i of the input operand x have to be converted from the normal domain to the order domain so that they can be fed into the SecMult gadgets at line 3, 5, and 7. In addition, as explained in the last section, the RefreshMasks gadget operates in the normal domain, and therefore its output masks have to be converted from the normal domain to the order domain because they are used as input for the first SecMult gadget in line 3. The input of RefreshMasks is the output of the first SecSquare gadget in line 1, which would normally be given in the order domain. However, instead of converting the SecSquare output from order domain to normal domain, we implemented a second SecSquare gadget that operates fully in the normal domain, i.e. both its inputs and outputs are elements of $\mathbb{F}_{2^8}$ and not orders.

A second aspect we have not discussed until now concerns the actual implementation of the gadgets. For reasons of efficiency, the SecMult gadget is "duplicated" and actually performs two masked multiplications instead of one, which means the `SECMULT` macro in Listing 7.2 consists of twice as many `add`, `ldrb`, and `eor` instructions, namely eight instead of four. The concrete reason why this can be beneficial speed-wise can be found in the pipeline structure of ARM Cortex-M microcontrollers. Namely, on certain Cortex models, neighboring `load` and `store` instructions can pipeline their address and data phases, which enables them to execute in a single cycle instead of two. Therefore, it is potentially beneficial to have sequence of eight `ldrb` instructions instead of four. Besides SecMult, we also duplicated the SecSquare and SecPow16 gadget, which means they always perform two masked squarings and two masked 16th-power operations. This duplications propagate to further up the exponentiation, i.e. we compute the secure exponentiation "pair-wise" on two masked bytes of the state.

Besides the inversion in $\mathbb{F}_{2^8}$, and S-box operation also involves and affine transformation, which is a linear operation and can be performed share-by-share using a 256-byte look-up table. At the highest level, our implementation of the first-order masked SubBytes transformation consists of a simple loop that is iterated eight times and executes in each iteration a pair of two masked S-box operations (i.e. two inversions followed by two affine transformations). Each first-order masked S-box requires five bytes of randomness (four for the SecMult gadgets and one for RefreshMasks), which amounts to 80 random bytes for the complete SubBytes. On our target device, which is a VL Discovery development board equipped with a STM32F100RBT6B Cortex-M3 microcontroller clocked at 24 MHz [STM16], the first-order masked SubBytes transformation has an execution time of 2299 clock cycles. This cycle count includes the loading of random bytes from an array in RAM, but not the generation of the random bytes itself.

The other three round transformations are relatively easy to implement since they consist solely of linear operations and can be performed share-by-share, independently from each other. Our implementation of the MixColumns transformation follows the approach of Bertoni et al [Ber+02], which is very efficient on 32-bit platforms. The execution time (including full function-call overhead) of our first-order masked ShiftRows, MixColumns, and AddRoundKey transformation on Cortex-M3 is 37, 133, and 78 clock cycles, respectively, which is almost negligible in relation to masked SubBytes. A single round of the masked AES executes in 2552 clock cycles, and the full masked AES encryption has an execution time of 25413 cycles.

7.5.2 Results and Comparison

We give in Table 7.1 a comparison between our work and the previous implementations on a similar architecture. It is somewhat hard to make a direct comparison since even if the platforms are similar, the techniques used for the masking schemes are quite different and might not offer similar level of security neither in theory nor in practice.
The most straightforward scheme to compare to is naturally the Rivain-Prouff masking scheme studied in [GR17] since it is the starting point of our work. We can see that the techniques described in Sections 7.3 and 7.4 largely improve the masking scheme as the cycle count is almost divided by two. Besides, our implementation is significantly more compact with a code size improvement of over 30%. On the affine masking side, the ANSSI implementation is beaten on both metrics when considering first order security. The contenders are thus the bitsliced/fixsliced implementations offering better speed results than ours and that are capable to process two blocks in parallel. We can firstly observe that their code is larger. However, we should fairly assume that large implementations were unrolled and that they did not try optimize this metric. Regarding security, [Gro+19] and [AP20] are reducing to the bare minimum the amount of randomness required to mask the whole scheme (only 2 bits). While they used a formal verification tool to assess the security in the probing model, using such an aggressive technique is quite naturally raising concerns for the practical security of the scheme and would not fit in our theoretical framework. Also, their technique does not generalize to arbitrary orders. More importantly, all

the bitsliced/fixsliced schemes are actually leaking according to the recent analysis in [Bec+22]. This means that all those schemes need to be fixed before proper benchmarking can be performed. Since it is impossible to tell beforehand how large the performance penalty incurred by this fix will be, the cycle count reported do not offer the possibility to compare to our result.

Reference	Implementation details	Exec. time (cycles)	Code size (kB)	Theoretical security	Practical security
[GR17]	RP masking	49,329 (ARM7)	4.8	Probing	unknown
[SS17]	Bitsliced	17,500 (M4)	39.9[1]	Probing?	t-test (failed [Bec+22])
[Gro+19]	Bitsliced	6,800 (M4)	25.2	Probing	t-test (failed [Bec+22])
[AP20]	Fixsliced	6,200 (M4)	22 or 4[2]	Probing	t-test (failed [Bec+22])
ANSSI (compact)	Affine[3]	53,072 (M4)	5.5	Affine	Attacked ([BS20; MS21])
ANSSI (unrolled)	Affine[3]	29,920 (M4)	25.0	Affine	Attacked ([BS20; MS21])
Our work	RP masking	25,413 (M3)	3.2	SNI	t-test (passed)

TABLE 7.1: Comparison of masked AES implementations for ARM Cortex-M3/M4 microcontrollers. [1]Schwabe et al. size is given for a full AES-CTR implementation, no size for AES block was given in their paper.[2] [AP20] indicates 2 versions of similar performances but [Bec+22] does not report which one was used for the benchmarks.[3] See [Fum+10] for a discussion about the security of affine masking.

7.6 Practical Leakage Assessment

As shown in the work by Beckers et al. [Bec+22], there is an important and seldom explored discrepancy between the theoretical and practical aspects of the security pertaining masking schemes. Theoretical proofs have their importance, but many works lack a practical demonstration of their claims.

Taking that into consideration, in this paper we aim to not only have good theoretical guarantees but also apply a practical test of those guaranties. The leakage assessment test is not a definitive tool to test and qualify an implementation of a masking scheme. Nonetheless, such tests are an important sanity test and demonstrate that there are no obvious mistakes in the masking scheme and its implementation. In this section, we will briefly talk about the implementation techniques and the TVLA test.

For testing the implementation described in this paper, we have used the general Test Vector Leakage Assessment (TVLA) test. The general test compares power measurements on a device with a fixed key, across datasets with random and fixed plaintext. To evaluate the leakage, the two datasets are compared for significant statistical differences using Welch's t-test.

Welch's t-test, comparing the measured traces from two subsets, F and R is done by computing: X_F and X_R, which is the average of all traces in each subset; and S_F and S_R, as the sample standard deviation of all traces in each subset. Given that each measured trace is a vector of measured values across time, the average and standard deviation are also vectors over time. The t-value is then computed point-wise as:

$$\frac{X_F - X_R}{\sqrt{\frac{S_F^2}{N_F} + \frac{S_R^2}{N_R}}}$$

Welch's t-test returns a threshold, expressed as standard deviations, that corresponds to the confidence that the difference between the two sets is not a random event. It

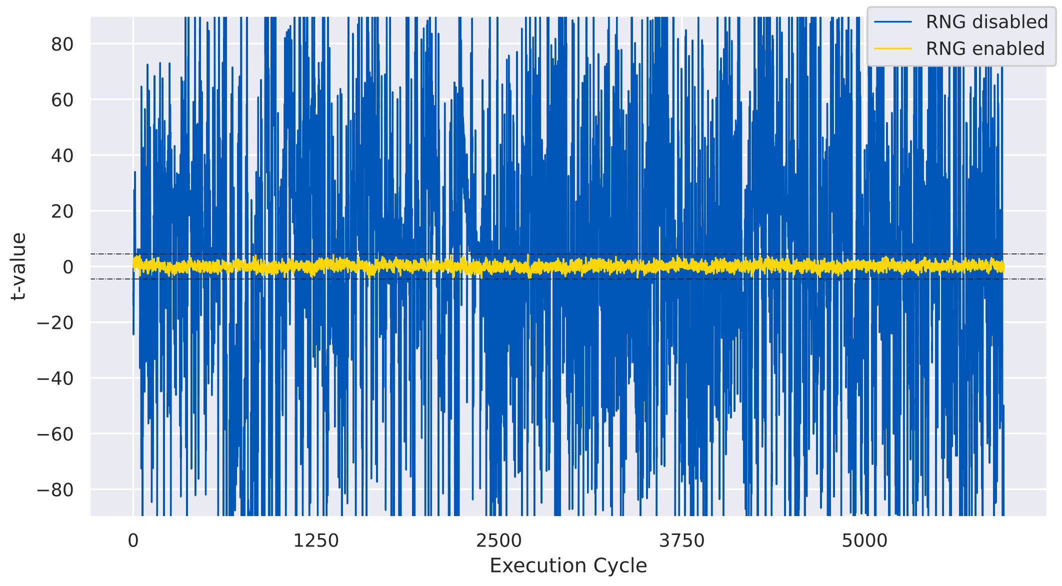

FIGURE 7.1: Graph showing the t-test result of the masked implementation. In lighter color, the normal execution, showing that the results stay within the $\pm 4.5\sigma$ boundary. In a darker color, the same experiment, but with one of the masks set to zero.

is usual to set this confidence value, in the context of leakage assessment, to 4.5 standard deviations –equivalent to 99.999% confidence. Furthermore, the test should be run twice on independent datasets, and if a point in both tests exceeds the $\pm 4.5\sigma$, the device is leaking data-related information.

For the test platform, we have used a ChipWhisperer-Lite 32-Bit, produced by NewAE Technology Inc. This is a single-board solution, fully open source side-channel (and power analysis and glitching) platform. The device features an integrated STM32F303 32-bit Arm Cortex M4 target and a 10-bit OpenADC scope. The CW-Lite was chosen due to its ease of use and low price point, together with the fact that it is bundled to a relevant target. Our tests were executed on the default build configuration for the ChipWhisperer, which sets the target's flash latency to zero. This eliminates waitstates, which might cause code-layout dependant behaviors.

One limitation of the ChipWhisperer-lite platform is the size of the scope buffer, being capable of measuring spans of around 5000 cycles in our tests (which corresponds to a bit more than two full AES rounds). This fact makes it more difficult to generate a measurement vector for a full round AES. On the other hand, due to the round structure of the cipher, measuring a full round should be enough demonstration of the behavior of the masked implementation, as long as it, as well, has a round structure.

The results of the TVLA test of our implementation are shown in Figure 7.1. Our implementation stays within the set boundaries of $\pm 4.5\sigma$, and shows leakage when one of the masks is set to zero. Notice that the y-axis is cropped, and the zero-mask, in some points, show a confidence in excess of 100σ. The experiment consists of 5000 pairs of power traces. It is not to say that our implementation is without shortcomings. Our implementation is optimized for first-order masking: our method can work for higher-order masking, but it is not trivial to implement, as special care will need to be taken with register allocation, for example.

Limitations of our Implementation. Our prototype implementation of the improved RP masking scheme serves mainly two purposes, namely (i) to evaluate the execution time so that we can demonstrate a significant speed-up compared to the work of Goudarzi and Rivain [GR17], and (ii) to be able to assess the security of our masking scheme in the real world by carrying out a t-test-based leakage evaluation. Even though the t-test confirms that our implementation is sound –in the sense that it does not contain an obvious flaw–, it still suffers from two shortcomings. First, our current implementation only supports first-order masking, which means it will most likely succumb under a second-order DPA, where an attacker tries to combine the leakages resulting from the manipulation of the two shares. However, we emphasize that the vulnerability to second-order DPA is not related to the optimizations we introduced in this paper since, in principle, any first-order masking scheme succumbs to a second-order DPA if the attacker is able to collect a sufficiently large number of traces. Hence, it is very likely that our masked AES implementation can be broken through a second-order DPA attack. However, we plan to tackle this issue together with the second shortcoming described below as part of our future work.

The second shortcoming of our current implementation is that we did not attempt to reduce or eliminate so-called micro-architectural leakages, which can be broadly defined as leakages introduced by certain micro-architectural effects and/or features like pipelining. It has been shown that, for example, the pipeline registers can cause a violation of the fundamental assumption that the processed shares leak separately and independently from each other so that there is no combined leakage. Even if a masking scheme has been correctly implemented, it can still leak secret information because of the micro-architecture, as was demonstrated in e.g. [CGD18; MMT20; MPW22; GOP21]. The literature contains a few techniques to reduce micro-architectural leakage, but getting fully rid of it requires detailed information about the concrete implementation of the micro-architecture; in the ideal case the HDL description of the target processor is available. Unfortunately, since the STM32F100RBT6B Cortex-M3 is a proprietary microcontroller, its HDL description is not publicly available. In our future work, we will therefore switch our target platform from ARM to RISC-V and we plan to develop first and higher-order masked AES implementations with countermeasures against micro-architectural leakages.

7.7 Conclusions

The masking scheme of Rivain and Prouff, introduced roughly 12 years ago, has attracted a lot of interest in the cryptographic research community because it stands on a solid theoretical foundation and supports arbitrary masking orders. Unfortunately, it has turned out that these features come at the expense of relatively poor performance; for example, currently the best implementation of a first-order protected AES has an execution time of almost 50,000 clock cycles on a 32-bit ARM microcontroller, which is prohibitively expensive for many applications. We demonstrated in this paper that the masking scheme of Rivain and Prouff can be made much faster, namely by almost a factor of two in comparison with the, to this date, best implementation in the literature. We achieved this speed-up by (i) an optimization of the multiplication in $\mathbb{F}_{2^8}$ that allows one to overcome the special treatment of 0-operands and avoid branch-and-compare instructions, and (ii) a new exponentiation technique that reduces the overall number of table look-ups. Our masking scheme is not only (almost) twice as fast as the original Rivain-Prouff scheme, it also satisfies strong non-interference, which is a stronger notion of probing security. In addition, our masked

AES does not rely solely on theoretical security properties since we performed a test-vector-based leakage evaluation as a “sanity test” to confirm our implementation is not flawed. As part of our future work, we plan to develop higher-order masked AES implementations for the RISC-V platform with integrated countermeasures against micro-architectural leakages.

Bibliography

[Abe+14] Farzaneh Abed, Eik List, Stefan Lucks, and Jakob Wenzel. “Differential Cryptanalysis of Round-Reduced Simon and Speck”. In: *FSE 2014*. Vol. 8540. LNCS. Springer, 2014, pp. 525–545.

[Aes] *Advanced Encryption Standard (AES)*. National Institute of Standards and Technology (NIST), FIPS PUB 197, U.S. Department of Commerce. 2001-11.

[AJN14] Jean-Philippe Aumasson, Philipp Jovanovic, and Samuel Neves. “NORX: Parallel and Scalable AEAD”. In: *ESORICS 2014*. Ed. by Miroslaw Kutylowski and Jaideep Vaidya. Vol. 8713. Lecture Notes in Computer Science. Springer, 2014, pp. 19–36. ISBN: 978-3-319-11211-4. DOI: 10.1007/978-3-319-11212-1_2. URL: http://dx.doi.org/10.1007/978-3-319-11212-1_2.

[AJN16] Jean-Philippe Aumasson, Philipp Jovanovic, and Samuel Neves. *NORX v3.0*. Specification, available for download at http://github.com/norx/resources/raw/master/specs/norxv30.pdf. 2016.

[AK18] Ralph Ankele and Stefan Kölbl. “Mind the Gap - A Closer Look at the Security of Block Ciphers against Differential Cryptanalysis”. In: *Selected Areas in Cryptography - SAC 2018 - 25th International Conference, Calgary, AB, Canada, August 15-17, 2018, Revised Selected Papers*. Ed. by Carlos Cid and Michael J. Jacobson Jr. Vol. 11349. Lecture Notes in Computer Science. Springer, 2018, pp. 163–190. DOI: 10.1007/978-3-030-10970-7_8. URL: https://doi.org/10.1007/978-3-030-10970-7_8.

[Ale+22] Biryukov Alex, Luan Cardoso dos Santos, Je Sen Teh, Aleksei Udovenko, and Vesselin Velichkov. “Meet-in-the-Filter and Dynamic Counting with Applications to Speck”. Cryptology ePrint Archive, Paper 2022/673. In submission. 2022. URL: https://eprint.iacr.org/2022/673/.

[AP20] Alexandre Adomnicai and Thomas Peyrin. “Fixslicing AES-like Ciphers: New bitsliced AES speed records on ARM-Cortex M and RISC-V”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2021.1 (2020), pp. 402425. DOI: 10.46586/tches.v2021.i1.402-425. URL: https://tches.iacr.org/index.php/TCHES/article/view/8739.

[Ard12] Arduino AG. *Arduino Due*. Specification, available online at http://www.arduino.cc/en/Main/ArduinoBoardDue. 2012.

[Arm16] Arm Limited. *ARM Cortex-M3 Processor Technical Reference Manual, Revision r2p1*. Available for download at http://developer.arm.com/documentation/100165/latest. 2016.

[Arm21] Arm Limited. *ARMv7-M Architecture Reference Manual, Issue E.e*. Available for download at http://developer.arm.com/documentation/ddi0403/latest. 2021.

[Bal+13] Josep Balasch, Baris Ege, Thomas Eisenbarth, Benoît Gérard, Zheng Gong, Tim Güneysu, Stefan Heyse, Stéphanie Kerckhof, François Koeune, Thomas Plos, Thomas Pöppelmann, Francesco Regazzoni, François-Xavier Standaert, Gilles Van Assche, Ronny Van Keer, Loïc van Oldeneel tot Oldenzeel, and Ingo von Maurich. "Compact Implementation and Performance Evaluation of Hash Functions in ATtiny Devices". In: *Smart Card Research and Advanced Applications — CARDIS 2012*. Ed. by Stefan Mangard. Vol. 7771. Lecture Notes in Computer Science. Springer Verlag, 2013, pp. 158–172. ISBN: 978-3-642-37287-2.

[Bao+21] Zhenzhen Bao, Jian Guo, Meicheng Liu, Li Ma, and Yi Tu. *Conditional Differential-Neural Cryptanalysis*. Cryptology ePrint Archive, Report 2021/719. 2021.

[BDG16] Alex Biryukov, Daniel Dinu, and Johann Großschädl. "Correlation Power Analysis of Lightweight Block Ciphers: From Theory to Practice". In: *International Conference on Applied Cryptography and Network Security – ACNS 2016*. Vol. 9696. Lecture Notes in Computer Science. Springer, 2016, pp. 537–557.

[Bea+13] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. *The SIMON and SPECK Families of Lightweight Block Ciphers*. Cryptology ePrint Archive, Report 2013/404. 2013.

[Bec+22] Arthur Beckers, Lennert Wouters, Benedikt Gierlichs, Bart Preneel, and Ingrid Verbauwhede. *Provable Secure Software Masking in the Real-World*. https://www.esat.kuleuven.be/cosic/publications/article-3461.pdf. 2022.

[Bee15] Daniel Beer. *MSPDebug: Debugging tool for MSP430 MCUs*. Available online at http://mspdebug.sourceforge.net. 2015.

[Bei+19] Christof Beierle, Alex Biryukov, Luan Cardoso dos Santos, Johann Großschädl, Léo Perrin, Aleksei Udovenko, Vesselin Velichkov, and Qingju Wang. *Schwaemm and Esch: Lightweight Authenticated Encryption and Hashing using the Sparkle Permutation Family*. https://www.cryptolux.org/index.php/Sparkle. 2019.

[Bei+20a] Christof Beierle, Alex Biryukov, Luan Cardoso dos Santos, Johann Großschädl, Léo Perrin, Aleksei Udovenko, Vesselin Velichkov, and Qingju Wang. "Alzette: A 64-Bit ARX-box". In: *Advances in Cryptology – CRYPTO 2020*. Ed. by Daniele Micciancio and Thomas Ristenpart. Cham: Springer International Publishing, 2020, pp. 419–448. ISBN: 978-3-030-56877-1.

[Bei+20b] Christof Beierle, Alex Biryukov, Luan Cardoso dos Santos, Johann Großschädl, Léo Perrin, Aleksei Udovenko, Vesselin Velichkov, and Qingju Wang. "Alzette: A 64-Bit ARX-box - (Feat. CRAX and TRAX)". In: *CRYPTO 2020*. Ed. by Daniele Micciancio and Thomas Ristenpart. Vol. 12172. LNCS. Springer, 2020, pp. 419–448. DOI: 10.1007/978-3-030-56877-1_15. URL: https://doi.org/10.1007/978-3-030-56877-1_15.

[Bei+20c] Christof Beierle, Alex Biryukov, Luan Cardoso dos Santos, Johann Großschädl, Léo Perrin, Aleksei Udovenko, Vesselin Velichkov, and Qingju Wang. *Lightweight AEAD and Hashing using the Sparkle Permutation Family*. 2020-06.

[Bei+20d] Christof Beierle, Alex Biryukov, Luan Cardoso dos Santos, Johann Großschädl, Léo Perrin, Aleksei Udovenko, Vesselin Velichkov, and Qingju Wang. "Lightweight AEAD and hashing using the sparkle permutation family". In: *IACR Transactions on Symmetric Cryptology* (2020), pp. 208–261.

[Ben+21] Adrien Benamira, David Gérault, Thomas Peyrin, and Quan Quan Tan. "A Deeper Look at Machine Learning-Based Cryptanalysis". In: *EUROCRYPT 2021*. Vol. 12696. LNCS. Springer, 2021, pp. 805–835.

[Ber+a] Daniel J. Bernstein, Stefan Kölbl, Stefan Lucks, Pedro Maat Costa Massolino, Florian Mendel, Kashif Nawaz, Tobias Schneider, Peter Schwabe, François-Xavier Standaert, Yosuke Todo, and Benoît Viguier. "Gimli : A Cross-Platform Permutation". In: pp. 299–320. DOI: 10.1007/978-3-319-66787-4_15.

[Ber+b] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. *CAESAR submission: Ketje v2, 2016.*

[Ber+02] Guido Bertoni, Luca Breveglieri, Pasqualina Fragneto, Marco Macchetti, and Stefano Marchesin. "Efficient Software Implementation of AES on 32-Bit Platforms". In: *Cryptographic Hardware and Embedded Systems — CHES 2002*. Ed. by Burton S. Kaliski, Çetin Kaya Koç, and Christof Paar. Vol. 2523. Lecture Notes in Computer Science. Springer, 2002, pp. 159–171.

[Ber08] Daniel J. Bernstein. "The Salsa20 Family of Stream Ciphers". In: *New Stream Cipher Designs – The eSTREAM Finalists*. Ed. by Matthew J. Robshaw and Olivier Billet. Vol. 4986. Lecture Notes in Computer Science. Springer Verlag, 2008, pp. 84–97. ISBN: 978-3-540-68350-6.

[Ber+11a] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. "Duplexing the Sponge: Single-Pass Authenticated Encryption and Other Applications". In: *Selected Areas in Cryptography - 18th International Workshop, SAC 2011, Toronto, ON, Canada, August 11-12, 2011, Revised Selected Papers*. 2011, pp. 320–337. DOI: 10.1007/978-3-642-28496-0_19. URL: https://doi.org/10.1007/978-3-642-28496-0_19.

[Ber+11b] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. *Cryptographic Sponge Functions*. Available for download at http://keccak.team/files/CSF-0.1.pdf. 2011.

[Ber+11c] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. *Cryptographic sponge functions*. Available at https://keccak.team/files/CSF-0.1.pdf. 2011.

[Ber+11d] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. "Duplexing the Sponge: Single-Pass Authenticated Encryption and Other Applications". In: *Selected Areas in Cryptography — SAC 2011*. Ed. by Ali Miri and Serge Vaudenay. Vol. 7118. Lecture Notes in Computer Science. Springer Verlag, 2011, pp. 320–337. ISBN: 978-3-642-28495-3.

[Ber+11e] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. *The Keccak Reference, Version 3.0*. Available for download at http://keccak.team/files/Keccak-reference-3.0.pdf. 2011.

[Ber+12] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. "Permutation-Based Encryption, Authentication and Authenticated Encryption". In: *Record of the 1st ECRYPT II Workshop on New Directions in Authenticated Encryption (DIAC 2012)*. 2012, pp. 159–170.

[Ber+16] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. *CAESAR submission: Ketje v2*. Submission to CAESAR, available via https://competitions.cr.yp.to/round3/ketjev2.pdf. 2016.

[Ber+17a] Daniel J. Bernstein, Stefan Kölbl, Stefan Lucks, Pedro M. Massolino, Florian Mendel, Kashif Nawaz, Tobias Schneider, Peter Schwabe, François-Xavier Standaert, Yosuke Todo, and Benoît Viguier. "Gimli: A Cross-Platform Permutation". In: *Cryptographic Hardware and Embedded Systems — CHES 2017*. Ed. by Wieland Fischer and Naofumi Homma. Vol. 10529. Lecture Notes in Computer Science. Springer Verlag, 2017, pp. 299–320. ISBN: 978-3-319-66786-7.

[Ber+17b] Daniel J. Bernstein, Stefan Kölbl, Stefan Lucks, Pedro Maat Costa Massolino, Florian Mendel, Kashif Nawaz, Tobias Schneider, Peter Schwabe, François-Xavier Standaert, Yosuke Todo, and Benoît Viguier. "Gimli : A Cross-Platform Permutation". In: *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*. Ed. by Wieland Fischer and Naofumi Homma. Vol. 10529. Lecture Notes in Computer Science. Springer, 2017, pp. 299–320.

[Ber+17c] Guido Bertoni, Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. "Farfalle: Parallel Permutation-Based Cryptography". In: *IACR Transactions on Symmetric Cryptology* 2017.4 (2017-12), pp. 1–38.

[Ber+18] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, Ronny Van Keer, and Benoît Viguier. "KangarooTwelve: Fast Hashing Based on Keccak-p". In: *Applied Cryptography and Network Security — ACNS 2018*. Ed. by Bart Preneel and Frederik Vercauteren. Vol. 10892. Lecture Notes in Computer Science. Springer Verlag, 2018, pp. 400–418. ISBN: 978-3-319-93386-3.

[Bir+22] Alex Biryukov, Luan Cardoso dos Santos, Daniel Feher, Vesselin Velichkov, and Giuseppe Vitto. "Automated Truncation of Differential Trails and Trail Clustering in ARX". In: *Selected Areas in Cryptography*. Ed. by Riham AlTawy and Andreas Hülsing. Cham: Springer International Publishing, 2022, pp. 286–307. ISBN: 978-3-030-99277-4.

[BL09] Daniel J. Bernstein and Tanja Lange. *eBACS: ECRYPT Benchmarking of Cryptographic Systems*. Available online at http://bench.cr.yp.to. 2009.

[BR00] Mihir Bellare and Phillip Rogaway. "Encode-Then-Encipher Encryption: How to Exploit Nonces or Redundancy in Plaintexts for Efficient Cryptography". In: *Advances in Cryptology — ASIACRYPT 2000*. Ed. by Tatsuaki Okamoto. Vol. 1976. Lecture Notes in Computer Science. Springer Verlag, 2000, pp. 317–330. ISBN: 3-540-41404-5.

[BRV14] Alex Biryukov, Arnab Roy, and Vesselin Velichkov. "Differential Analysis of Block Ciphers SIMON and SPECK". In: *FSE 2014*. Ed. by Carlos Cid and Christian Rechberger. Vol. 8540. Lecture Notes in Computer Science. Springer, 2014, pp. 546–570. ISBN: 978-3-662-46705-3. DOI: 10.1007/978-3-662-46706-0_28. URL: http://dx.doi.org/10.1007/978-3-662-46706-0_28.

[BS20] Olivier Bronchain and François-Xavier Standaert. "Side-Channel Countermeasures' Dissection and the Limits of Closed Source Security Evaluations". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020.2 (2020), pp. 1–25. DOI: 10.13154/tches.v2020.i2.1-25. URL: https://doi.org/10.13154/tches.v2020.i2.1-25.

[BS91] Eli Biham and Adi Shamir. "Differential Cryptanalysis of DES-like Cryptosystems". In: *J. Cryptology* 4.1 (1991), pp. 3–72.

[Bur14] Elie Bursztein. *Speeding up and Strengthening HTTPS Connections for Chrome on Android.* Google Security Blog, available online at `http://security.googleblog.com/2014/04/speeding-up-and-strengthening-https.html`. 2014.

[BV14] Alex Biryukov and Vesselin Velichkov. "Automatic Search for Differential Trails in ARX Ciphers". In: *CT-RSA 2014.* Ed. by Josh Benaloh. Vol. 8366. Lecture Notes in Computer Science. Springer, 2014, pp. 227–250. ISBN: 978-3-319-04851-2. DOI: `10.1007/978-3-319-04852-9_12`. URL: `http://dx.doi.org/10.1007/978-3-319-04852-9_12`.

[BVC16] Alex Biryukov, Vesselin Velichkov, and Yann Le Corre. "Automatic Search for the Best Trails in ARX: Application to Block Cipher Speck". In: *FSE 2016.* Vol. 9783. LNCS. Springer, 2016, pp. 289–310.

[Car+18] Matthew R Carter, Raghurama R Velagala, John Pham, and Jens-Peter Kaps. *eXtended eXternal Benchmarking eXtension (XXBX).* IEEE International Symposium on Hardware Oriented Security and Trust (HOST) 2018. 2018.

[CDG19] Hao Cheng, Daniel Dinu, and Johann Großschädl. "Efficient Implementation of the SHA-512 Hash Function for 8-Bit AVR Microcontrollers". In: *Innovative Security Solutions for Information Technology and Communications - 11th International Conference, SecITC 2018, Bucharest, Romania, November 8-9, 2018, Revised Selected Papers.* Ed. by Jean-Louis Lanet and Cristian Toma. Vol. 11359. Lecture Notes in Computer Science. Springer Verlag, 2019, pp. 273–287.

[CGD18] Yann Le Corre, Johann Großschädl, and Daniel Dinu. "Micro-architectural Power Simulator for Leakage Assessment of Cryptographic Software on ARM Cortex-M3 Processors". In: *Constructive Side-Channel Analysis and Secure Design - 9th International Workshop, COSADE 2018, Singapore, April 23-24, 2018, Proceedings.* Ed. by Junfeng Fan and Benedikt Gierlichs. Vol. 10815. Lecture Notes in Computer Science. Springer, 2018, pp. 82–98. DOI: `10.1007/978-3-319-89641-0\_5`. URL: `https://doi.org/10.1007/978-3-319-89641-0\_5`.

[CGV14] Jean-Sébastien Coron, Johann Großschädl, and Praveen Kumar Vadnala. "Secure Conversion between Boolean and Arithmetic Masking of Any Order". In: *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings.* Ed. by Lejla Batina and Matthew Robshaw. Vol. 8731. Lecture Notes in Computer Science. Springer, 2014, pp. 188–205.

[Cha+18] Avik Chakraborti, Nilanjan Datta, Mridul Nandi, and Kan Yasuda. "Beetle Family of Lightweight and Secure Authenticated Encryption Ciphers". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2018.2 (2018-05), pp. 218–241.

[Cor+14] Jean-Sébastien Coron, Emmanuel Prouff, Matthieu Rivain, and Thomas Roche. "Higher-Order Side Channel Security and Mask Refreshing". In: *Fast Software Encryption.* Ed. by Shiho Moriai. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 410–424. ISBN: 978-3-662-43933-3.

[Cor+15] Jean-Sébastien Coron, Johann Großschädl, Mehdi Tibouchi, and Praveen Kumar Vadnala. "Conversion from Arithmetic to Boolean Masking with Logarithmic Complexity". In: *Fast Software Encryption - 22nd International Workshop, FSE 2015, Istanbul, Turkey, March 8-11, 2015, Revised Selected Papers.* Ed. by Gregor Leander. Vol. 9054. Lecture Notes in Computer Science. Springer, 2015, pp. 130–149.

[Cry16] CryptoLUX Team. *FELICS: Fair Evaluation of Lightweight Cryptographic Systems*. Available online at http://www.cryptolux.org/index.php/FELICS. 2016.

[Dae+18] Joan Daemen, Seth Hoffert, Gilles Van Assche, and Ronny Van Keer. “The Design of Xoodoo and Xoofff”. In: *IACR Transactions on Symmetric Cryptology* 2018.4 (2018-12), pp. 1–38.

[Dae+20] Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. “Xoodyak, a Lightweight cryptographic Scheme”. In: *IACR Transactions on Symmetric Cryptology* 2020.S1 (2020-06), pp. 60–87.

[Des] *Data Encryption Standard*. National Bureau of Standards, NBS FIPS PUB 46, U.S. Department of Commerce. 1977-01.

[DFJ13] Patrick Derbez, Pierre-Alain Fouque, and Jérémy Jean. “Improved Key Recovery Attacks on Reduced-Round AES in the Single-Key Setting”. In: *EUROCRYPT 2013*. Vol. 7881. LNCS. Springer, 2013, pp. 371–387.

[Din+] Daniel Dinu, Léo Perrin, Aleksei Udovenko, Vesselin Velichkov, Johann Großschädl, and Alex Biryukov. “Design Strategies for ARX with Provable Bounds: Sparx and LAX”. In: pp. 484–513. DOI: 10.1007/978-3-662-53887-6_18.

[Din14] Itai Dinur. “Improved Differential Cryptanalysis of Round-Reduced Speck”. In: *SAC 2014*. Vol. 8781. LNCS. Springer, 2014, pp. 147–164.

[Din+15a] Daniel Dinu, Alex Biryukov, Johann Großschädl, Dmitry Khovratovich, YL Corre, and Léo Perrin. “Felics–fair evaluation of lightweight cryptographic systems”. In: *NIST Workshop on Lightweight Cryptography*. Vol. 128. 2015.

[Din+15b] Daniel Dinu, Yann Le Corre, Dmitry Khovratovich, Leo Perrin, Johann Großschädl, and Alex Biryukov. *Triathlon of Lightweight Block Ciphers for the Internet of Things*. Cryptology ePrint Archive, Report 2015/209. https://eprint.iacr.org/2015/209. 2015.

[DKS10] Orr Dunkelman, Nathan Keller, and Adi Shamir. “Improved Single-Key Attacks on 8-Round AES-192 and AES-256”. In: *ASIACRYPT 2010*. Vol. 6477. LNCS. Springer, 2010, pp. 158–176.

[Dob+14] C Dobraunig, M Eichlseder, Florian Mendel, and M Schläffer. “ASCON v1, submission to the CAESAR competition”. In: *CAESAR First Round Submission, March* (2014).

[Dob+16] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. *Ascon v1.2*. Submission to CAESAR, available via https://competitions.cr.yp.to/round3/asconv12.pdf. 2016.

[Dob+20] Christoph Dobraunig, Maria Eichlseder, Stefan Mangard, Florian Mendel, Bart Mennink, Robert Primas, and Thomas Unterluggauer. “Isap v2.0”. In: *IACR Transactions on Symmetric Cryptology* 2020.S1 (2020-06), pp. 390–416.

[Dob+21] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. “Ascon v1.2: Lightweight Authenticated Encryption and Hashing”. In: *Journal of Cryptology* 34.3 (2021-07), p. 33.

[DR02] Joan Daemen and Vincent Rijmen. “Specification of Rijndael”. In: *The Design of Rijndael: AES — The Advanced Encryption Standard*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 31–51. ISBN: 978-3-662-04722-4. DOI: 10.1007/978-3-662-04722-4_3. URL: https://doi.org/10.1007/978-3-662-04722-4_3.

[DR06] Christophe De Cannière and Christian Rechberger. "Finding SHA-1 Characteristics: General Results and Applications". In: *ASIACRYPT*. Ed. by Xuejia Lai and Kefei Chen. Vol. 4284. Lecture Notes in Computer Science. Springer, 2006, pp. 1–20. ISBN: 3-540-49475-8.

[DSP07] Orr Dunkelman, Gautham Sekar, and Bart Preneel. "Improved Meet-in-the-Middle Attacks on Reduced-Round DES". In: *INDOCRYPT 2007*. Vol. 4859. LNCS. Springer, 2007, pp. 86–100.

[Fló+21] Antonio Flórez-Gutiérrez, Gaëtan Leurent, María Naya-Plasencia, Léo Perrin, André Schrottenloher, and Ferdinand Sibleyras. "Internal Symmetries and Linear Properties: Full-permutation Distinguishers and Improved Collisions on Gimli". In: *Journal of Cryptology* 34.4 (2021-10), p. 45.

[FPS17] Sebastian Faust, Clara Paglialonga, and Tobias Schneider. "Amortizing Randomness Complexity in Private Circuits". In: (2017-11), pp. 781–810. DOI: 10.1007/978-3-319-70694-8_27.

[Fu+16] Kai Fu, Meiqin Wang, Yinghua Guo, Siwei Sun, and Lei Hu. "MILP-Based Automatic Search Algorithms for Differential and Linear Trails for Speck". In: *FSE 2016*. Vol. 9783. LNCS. Springer, 2016, pp. 268–288.

[Fum+10] Guillaume Fumaroli, Ange Martinelli, Emmanuel Prouff, and Matthieu Rivain. "Affine Masking against Higher-Order Side Channel Analysis". In: *IACR Cryptol. ePrint Arch.* 2010 (2010), p. 523.

[Gla07] Brian R. Gladman. *A Specification for Rijndael, the AES Algorithm*. Technical report, available for download at http://ccgi.gladman.plus.com/oldsite/cryptography_technology/rijndael/aes.spec.v316.pdf. 2007.

[Goh19] Aron Gohr. "Improving Attacks on Round-Reduced Speck32/64 Using Deep Learning". In: *CRYPTO 2019*. Vol. 11693. LNCS. Springer, 2019, pp. 150–179.

[GOP21] Si Gao, Elisabeth Oswald, and Dan Page. "Reverse Engineering the Micro-Architectural Leakage Features of a Commercial Processor". In: *IACR Cryptol. ePrint Arch.* (2021), p. 794. URL: https://eprint.iacr.org/2021/794.

[GR17] Dahmun Goudarzi and Matthieu Rivain. "How Fast Can Higher-Order Masking Be in Software?" In: *Advances in Cryptology — EUROCRYPT 2017*. Ed. by Jean-Sébastien Coron and Jesper B. Nielsen. Vol. 10210. Lecture Notes in Computer Science. http://ia.cr/2016/264. Springer Verlag, 2017, pp. 567–597.

[Gro+19] Hannes Gross, Ko Stoffelen, Lauren De Meyer, Martin Krenn, and Stefan Mangard. "First-Order Masking with Only Two Random Bits". In: *Proceedings of ACM Workshop on Theory of Implementation Security Workshop*. TIS'19. London, United Kingdom: Association for Computing Machinery, 2019, pp. 1023. ISBN: 9781450368278. DOI: 10.1145/3338467.3358950. URL: https://doi.org/10.1145/3338467.3358950.

[Hir16] Shoichi Hirose. "Sequential Hashing with Minimum Padding". In: *NIST Workshop on Lightweight Cryptography 2016*. National Institute of Standards and Technology (NIST). 2016.

[HW19] Mingjiang Huang and Liming Wang. "Automatic Tool for Searching for Differential Characteristics in ARX Ciphers and Applications". In: *INDOCRYPT 2019*. Vol. 11898. LNCS. Springer, 2019, pp. 115–138.

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. "Private Circuits: Securing Hardware against Probing Attacks". In: *Advances in Cryptology -*

CRYPTO 2003. Ed. by Dan Boneh. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 463–481. ISBN: 978-3-540-45146-4.

[JS17] Anthony Journault and François-Xavier Standaert. "Very High Order Masking: Efficient Implementation and Security Evaluation". In: *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*. Ed. by Wieland Fischer and Naofumi Homma. Vol. 10529. Lecture Notes in Computer Science. Springer, 2017, pp. 623–643. DOI: 10.1007/978-3-319-66787-4_30. URL: https://doi.org/10.1007/978-3-319-66787-4_30.

[Knu94] Lars R. Knudsen. "Truncated and Higher Order Differentials". In: *FSE 1994*. Vol. 1008. LNCS. Springer, 1994, pp. 196–211.

[KY01] Jonathan Katz and Moti Yung. "Unforgeable Encryption and Chosen Ciphertext Secure Modes of Operation". In: *Fast Software Encryption — FSE 2000*. Ed. by Bruce Schneier. Vol. 1978. Lecture Notes in Computer Science. Springer Verlag, 2001, pp. 284–299. ISBN: 3-540-41728-1.

[Lee+18] HoChang Lee, Seojin Kim, HyungChul Kang, Deukjo Hong, Jaechul Sung, and Seokhie Hong. "Calculating the Approximate Probability of Differentials for ARX-Based Cipher Using SAT Solver". In: *Journal of the Korea Institute of Information Security & Cryptology* 28.1 (2018), pp. 15–24.

[Leu12] Gaëtan Leurent. "Analysis of Differential Attacks in ARX Constructions". In: *ASIACRYPT 2012*. Ed. by Xiaoyun Wang and Kazue Sako. Vol. 7658. Lecture Notes in Computer Science. Springer, 2012, pp. 226–243. ISBN: 978-3-642-34960-7. DOI: 10.1007/978-3-642-34961-4_15. URL: http://dx.doi.org/10.1007/978-3-642-34961-4_15.

[Leu13] Gaëtan Leurent. "Construction of Differential Characteristics in ARX Designs Application to Skein". In: *CRYPTO 2013*. Ed. by Ran Canetti and Juan A. Garay. Vol. 8042. Lecture Notes in Computer Science. Springer, 2013, pp. 241–258. ISBN: 978-3-642-40040-7. DOI: 10.1007/978-3-642-40041-4_14. URL: http://dx.doi.org/10.1007/978-3-642-40041-4_14.

[LIM21] Fukang Liu, Takanori Isobe, and Willi Meier. "Cryptanalysis of Full LowMC and LowMC-M with Algebraic Techniques". In: *CRYPTO 2021*. Vol. 12827. LNCS. Springer, 2021, pp. 368–401.

[Liu+19] Zhengbin Liu, Yongqiang Li, Lin Jiao, and Mingsheng Wang. *A new method for Searching Optimal Differential and Linear Trails in ARX Ciphers*. Cryptology ePrint Archive, Report 2019/1438. 2019.

[LM01] Helger Lipmaa and Shiho Moriai. "Efficient Algorithms for Computing Differential Properties of Addition". In: *FSE*. Vol. 2355. LNCS. Springer, 2001, pp. 336–350.

[LM91] Xuejia Lai and James L. Massey. "Markov Ciphers and Differentail Cryptanalysis". In: *EUROCRYPT*. Ed. by Donald W. Davies. Vol. 547. Lecture Notes in Computer Science. Springer, 1991, pp. 17–38. ISBN: 3-540-54620-0.

[LP20] 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: *USENIX*. 2020.

[Mat94] Mitsuru Matsui. "On Correlation Between the Order of S-boxes and the Strength of DES". In: *EUROCRYPT*. Ed. by Alfredo De Santis. Vol. 950.

Lecture Notes in Computer Science. Springer, 1994, pp. 366–375. ISBN: 3-540-60176-7.

[Mic11] Microchip Technology Inc. *8-bit Atmel Microcontroller with 128KBytes In-System Programmable Flash: ATmega128, ATmega128L*. Available for download at http://ww1.microchip.com/downloads/en/DeviceDoc/doc2467.pdf. 2011.

[Mic20] Microchip Technology Inc. *AVR Instruction Set Manual*. Available for download at http://ww1.microchip.com/downloads/en/DeviceDoc/AVR-Instruction-Set-Manual-DS40002198A.pdf. 2020.

[MMT20] Lauren De Meyer, Elke De Mulder, and Michael Tunstall. "On the Effect of the (Micro)Architecture on the Development of Side-Channel Resistant Software". In: *IACR Cryptol. ePrint Arch.* (2020), p. 1297. URL: https://eprint.iacr.org/2020/1297.

[MNS11] Florian Mendel, Tomislav Nad, and Martin Schläffer. "Finding SHA-2 Characteristics: Searching through a Minefield of Contradictions". In: *ASIACRYPT*. Ed. by Dong Hoon Lee and Xiaoyun Wang. Vol. 7073. Lecture Notes in Computer Science. Springer, 2011, pp. 288–307. ISBN: 978-3-642-25384-3.

[Mor20] Mordor Intelligence, Inc. *8-bit Microcontroller Market – Growth, Trends, and Forecast (2020–2025)*. Available for download at http://www.mordorintelligence.com/industry-reports/8-bit-microcontroller-market-industry. 2020.

[Mou+11] Nicky Mouha, Qingju Wang, Dawu Gu, and Bart Preneel. "Differential and Linear Cryptanalysis Using Mixed-Integer Linear Programming". In: *Inscrypt 2011*. Ed. by Chuankun Wu, Moti Yung, and Dongdai Lin. Vol. 7537. Lecture Notes in Computer Science. Springer, 2011, pp. 57–76. ISBN: 978-3-642-34703-0. DOI: 10.1007/978-3-642-34704-7_5. URL: http://dx.doi.org/10.1007/978-3-642-34704-7_5.

[MP13] Nicky Mouha and Bart Preneel. *Towards Finding Optimal Differential Characteristics for ARX: Application to Salsa20*. Cryptology ePrint Archive, Report 2013/328. 2013.

[MPW22] Ben Marshall, Dan Page, and James Webb. "MIRACLE: MIcRo-ArChitectural Leakage Evaluation A study of micro-architectural power leakage across many devices". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022.1 (2022), pp. 175–220. DOI: 10.46586/tches.v2022.i1.175-220. URL: https://doi.org/10.46586/tches.v2022.i1.175-220.

[MRV15] Bart Mennink, Reza Reyhanitabar, and Damian Vizár. "Security of Full-State Keyed Sponge and Duplex: Applications to Authenticated Encryption". In: *Advances in Cryptology — ASIACRYPT 2015*. Ed. by Tetsu Iwata and Jung H. Cheon. Vol. 9453. Lecture Notes in Computer Science. Springer Verlag, 2015, pp. 465–489. ISBN: 978-3-662-48799-0.

[MS21] Loïc Masure and Rémi Strullu. "Side Channel Analysis against the ANSSI's protected AES implementation on ARM". In: *IACR Cryptol. ePrint Arch.* (2021), p. 592. URL: https://eprint.iacr.org/2021/592.

[MV04] David McGrew and John Viega. "The Galois/counter mode of operation (GCM)". In: *Submission to NIST Modes of Operation Process* 20 (2004).

[Nat15] National Institute of Standards and Technology (NIST). *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. FIPS Publication 202, available for download at http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf. Gaithersburg, MD, USA, 2015-08.

[Nat18] National Institute of Standards and Technology (NIST). *Submission Requirements and Evaluation Criteria for the Lightweight Cryptography Standardization Process.* Technical report, available for download at http://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/final-lwc-submission-requirements-august2018.pdf. 2018.

[Nat21] National Institute of Standards and Technology (NIST). *Status Report on the Second Round of the NIST Lightweight Cryptography Standardization Process.* Internal Report 8369, available for download at http://nvlpubs.nist.gov/nistpubs/ir/2021/NIST.IR.8369.pdf. Gaithersburg, MD, USA, 2021-07.

[Rad15] Radiant Insights, Inc. *Microcontroller Market Size, Share, Analysis Report 2020.* Available for download at http://www.radiantinsights.com/research/microcontroller-market/. 2015.

[Rog02] Phillip Rogaway. "Authenticated-Encryption with Associated-Data". In: *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS 2002).* Ed. by Vijayalakshmi Atluri. ACM Press, 2002, pp. 98–107. ISBN: 1-58113-612-9.

[RP10] Matthieu Rivain and Emmanuel Prouff. "Provably Secure Higher-Order Masking of AES". In: *Cryptographic Hardware and Embedded Systems, CHES 2010.* Ed. by Stefan Mangard and François-Xavier Standaert. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 413–427. ISBN: 978-3-642-15031-9.

[RST18] Christian Rechberger, Hadi Soleimany, and Tyge Tiessen. "Cryptanalysis of Low-Data Instances of Full LowMCv2". In: *IACR Trans. Symmetric Cryptol.* 2018.3 (2018), pp. 163–181.

[San+22] Luan Cardoso dos Santos, François Gérard, Johann Großschädl, and Lorenzo Spignoli. "Rivain-Prouff on Steroids: Faster and Stronger Masking of the AES". In submission. 2022.

[SG21] Luan Cardoso dos Santos and Johann Großschädl. "An Evaluation of the Multi-Platform Efficiency of Lightweight Cryptographic Permutations". In: *Innovative Security Solutions for Information Technology and Communications* (2021).

[SGB19] Luan Cardoso dos Santos, Johann Großschädl, and Alex Biryukov. "FELICS-AEAD: benchmarking of lightweight authenticated encryption algorithms". In: *International Conference on Smart Card Research and Advanced Applications.* Springer. 2019, pp. 216–233.

[Sha49] Claude E. Shannon. "Communication theory of secrecy systems". In: *Bell Syst. Tech. J.* 28.4 (1949), pp. 656–715. DOI: 10.1002/j.1538-7305.1949.tb00928.x. URL: https://doi.org/10.1002/j.1538-7305.1949.tb00928.x.

[SHY16] Ling Song, Zhangjie Huang, and Qianqian Yang. "Automatic Differential Analysis of ARX Block Ciphers with Application to SPECK and LEA". In: *ACISP 2016.* Vol. 9723. LNCS. Springer, 2016, pp. 379–394.

[SMG15] Tobias Schneider, Amir Moradi, and Tim Güneysu. "Arithmetic Addition over Boolean Masking - Towards First- and Second-Order Resistance in Hardware". In: *ACNS 2015.* Ed. by Tal Malkin, Vladimir Kolesnikov, Allison Bishop Lewko, and Michalis Polychronakis. Vol. 9092. LNCS. Springer, 2015, pp. 559–578.

[SS17] Peter Schwabe and Ko Stoffelen. "All the AES You Need on Cortex-M3 and M4". In: *Selected Areas in Cryptography – SAC 2016.* Ed. by

Roberto Avanzi and Howard Heys. Cham: Springer International Publishing, 2017, pp. 180–194. ISBN: 978-3-319-69453-5.

[Ste13] Marc Stevens. "New collision attacks on SHA-1 based on optimal joint local-collision analysis". In: *Advances in Cryptology–EUROCRYPT 2013*. Springer Berlin Heidelberg, 2013, pp. 245–261.

[Ste+17] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. "The First Collision for Full SHA-1". In: *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part I*. Ed. by Jonathan Katz and Hovav Shacham. Vol. 10401. Lecture Notes in Computer Science. Springer, 2017, pp. 570–596. DOI: `10.1007/978-3-319-63688-7\_19`. URL: `https://doi.org/10.1007/978-3-319-63688-7\_19`.

[STM16] STMicroelectronics. *STM32F100x4 STM32F100x6 STM32F100x8 STM32F100xB*. Data sheet, available for download at `http://www.st.com/resource/en/datasheet/stm32f100cb.pdf`. 2016.

[Sun+14] Siwei Sun, Lei Hu, Meiqin Wang, Peng Wang, Kexin Qiao, Xiaoshuang Ma, Danping Shi, Ling Song, and Kai Fu. *Towards Finding the Best Characteristics of Some Bit-oriented Block Ciphers and Automatic Enumeration of (Related-key) Differential and Linear Characteristics with Predefined Properties*. Cryptology ePrint Archive, Report 2014/747. `http://eprint.iacr.org/`. 2014.

[Sun+15] Siwei Sun, Lei Hu, Meiqin Wang, Qianqian Yang, Kexin Qiao, Xiaoshuang Ma, Ling Song, and Jinyong Shan. "Extending the Applicability of the Mixed-Integer Programming Technique in Automatic Differential Cryptanalysis". In: *Information Security - 18th International Conference, ISC 2015, Trondheim, Norway, September 9-11, 2015, Proceedings*. Ed. by Javier Lopez and Chris J. Mitchell. Vol. 9290. Lecture Notes in Computer Science. Springer, 2015, pp. 141–157. ISBN: 978-3-319-23317-8. DOI: `10.1007/978-3-319-23318-5_8`. URL: `http://dx.doi.org/10.1007/978-3-319-23318-5_8`.

[Tel17] Telefonaktiebolaget LM Ericsson. *Ericsson Mobility Report November 2017*. Available for download at `http://www.ericsson.com/assets/local/mobility-report/documents/2017/ericsson-mobility-report-november-2017.pdf`. 2017.

[TLP05] Ben L Titzer, Daniel K Lee, and Jens Palsberg. "Avrora: Scalable sensor network simulation with precise timing". In: *Information Processing in Sensor Networks, 2005. IPSN 2005. Fourth International Symposium on.* IEEE. 2005, pp. 477–482.

[Vel] Vesselin Velichkov. *YAARX: Yet Another Toolkit for Analysis of ARX Cryptographic Algorithms (2012–2018), Source code: `https://github.com/vesselinux/yaarx`, Documentation: `https://vesselinux.github.io/yaarx/index.html`*. Retrieved on January 2018.

[WBG10] Christian Wenzel-Benner and Jens Gräf. "XBX: eXternal Benchmarking eXtension for the SUPERCOP crypto benchmarking framework". In: *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer. 2010, pp. 294–305.

[Wu16] Hongjun Wu. "ACORN: a lightweight authenticated cipher (v3)". In: *Candidate for the CAESAR Competition. See also https://competitions. cr. yp. to/round3/acornv3. pdf* (2016).