SCHWAEMM and ESCH: Lightweight Authenticated Encryption and Hashing using the SPARKLE Permutation Family

Christof Beierle[1,2], Alex Biryukov[1], Luan Cardoso dos Santos[1], Johann Großschädl[1], Amir Moradi[2], Léo Perrin[3], Aein Rezaei Shahmirzadi[2], Aleksei Udovenko[1,4], Vesselin Velichkov[5], and Qingju Wang[1]

[1]DSC and SnT, University of Luxembourg, Luxembourg
[2]Ruhr University Bochum, Horst Görtz Institute for IT Security, Germany
[3]Inria, Paris, France
[4]CryptoExperts, Paris, France
[5]University of Edinburgh, U.K.

Version v1.2 (2021-05-17)

Corresponding submitter:

Prof. Dr. Alex Biryukov
Email: alex.biryukov@uni.lu
Phone: +352 466644-6793
University of Luxembourg
Maison du Nombre, 6, Avenue de la Fonte,
L–4364 Esch-sur-Alzette,
Luxembourg

Contact email for the whole SPARKLE group:
sparklegrupp@googlegroups.com

Homepage of SCHWAEMM, ESCH and SPARKLE:
https://sparkle-lwc.github.io/

Algorithms Specified in this Document

Type	Name	Internal state size (bytes)	Data block size (bytes)	Security level (bits)	Data limit (bytes)
Hash function	Esch256†	48	16	128	2^{132}
	Esch384	64	16	192	2^{196}
Extendable-output function (XOF)	XOEsch256	48	16	$\min\{128, t\}$	2^{132}
	XOEsch384	64	16	$\min\{192, t\}$	2^{196}
AEAD	Schwaemm128-128	32	16	120	2^{68}
	Schwaemm256-128†	48	32	120	2^{68}
	Schwaemm192-192	48	24	184	2^{68}
	Schwaemm256-256	64	32	248	2^{133}

† Primary instances.

We, the authors, faithfully declare that the algorithms presented in this document are, to the best of our knowledge, safe from all attacks currently known. We have not hidden any weakness in any of them.

Changelog. Version v$X.Y$ refers to the Y-th updated version of the pdf with regard to the algorithm specification X. Thus, differences in v$X.Y$ and v$X.Y'$ are only in the pdf, not in the actual algorithms. New and improved implementations might be provided.

- v1.0 to v1.1: Besides correcting minor typos, we did the following changes. We switched the primary member of the AEAD schemes from Schwaemm192-192 to Schwaemm256-128. We give a name to the ARX-box used in Sparkle, i.e., Alzette. We further added a clarification on how to map bitstrings to 32-bit words of the state in Sparkle. We also added new implementation results in Chapter 5.
- v1.1 to v1.2: Besides correcting minor typos, we did the following changes. First of all, our team is extended by Amir Moradi and Aein Rezaei Shahmirzadi. Moreover, we included the results of our latest publications [BBCdS+20a] and [BBCdS+20b] in this document. In particular, we included the extendable-output functions XOEsch256 and XOEsch384. Further, we could slightly improve the differential bound of Alzette. More precisely, we now have that the probability of the best 7-round differential trail is equal to 2^{-26}, which improves upon our previous result which only stated that this probability was at most 2^{-24}. We also expanded the division property analysis of Alzette.

Acknowledgements. The work of Christof Beierle was performed while he was at the University of Luxembourg and funded by the SnT CryptoLux RG budget. Luan Cardoso dos Santos is supported by the Luxembourg National Research Fund through grant PRIDE15/10621687/SPsquared. The work of Aleksei Udovenko is funded by the Fonds National de la Recherche Luxembourg (project reference 9037104). Part of the work by Vesselin Velichkov was performed while he was at the University of Luxembourg. The work of Qingju Wang is funded by the University of Luxembourg Internal Research Project (IRP) FDISC.

We thank Mridul Nandi for answering some questions about the Beetle mode of operation and also Benoît Cogliati for helping out with questions about provable security of variations of the Beetle mode.

The experiments presented in this paper were carried out using the HPC facilities of the University of Luxembourg [VBCG14] – see https://hpc.uni.lu.

Contents

Notations and Abbreviations

$\mathbb{F}_2$	The set $\{0, 1\}$
$\mathbb{F}_2^n$	The set of the bitstrings of length n
$\mathbb{F}_2^*$	The set of bitstrings of arbitrary length
n_b	number of branches
h_b	$\frac{n_b}{2}$
Word	An element of $\mathbb{F}_2^{32}$
Branch	A pair (x, y) of two words
$+$	Modular addition
$\oplus$	Exclusive or, also called "XOR"
$\|\|$	Concatenation of bitstrings
B^i	The i-times concatenation of the bitstring B with itself
&	The bitwise AND operation
$x \lll s$	The word x rotated left by s bits
$x \ggg s$	The word x rotated right by s bits
$x \ll s$	The word x shifted left by s bits
$x \gg s$	The word x shifted right by s bits
$\mathrm{hw}(x)$	The Hamming weight of the bitstring x
ϵ	A binary string of length 0
$\|E\|$	The number of elements in a set E or the length of a string in bit
$\Pr[\omega]$	The probability of an event ω
$\mathrm{I}, \mathcal{I}$	The identity mapping
IoT	Internet of Things
AEAD	Authenticated encryption with associated data
ARX	Add-Rotation-XOR
SPN	Substitution-Permutation Network
LTS	Long trail strategy
SIMD	Single Instruction Multiple Data

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.

Note. The Sparkle family of permutations together with the AEAD instances Schwaemm and the hash functions Esch is published in the Special Issue of *IACR Transactions on Symmetric Cryptology* dedicated to the second-round candidates of the NIST lightweight cryptography standardization process [BBCdS+20b].

1.1 What Is This Document?

In this document, we specify the cryptographic hash function family Esch and the authenticated encryption scheme Schwaemm, submitted to the NIST *lightweight cryptography standardization process*.

Together with the specification of the algorithms (Chapter 2), we provide a detailed design rationale that explains the choice of the overall structure and its internal components (Chapter 3). Further, we provide a detailed analysis of the security of our schemes with regard to state-of-the art attacks, and beyond (Chapter 4). Proper design rationale and security analysis are essential parts in a design proposal as they are necessary for other people to trust cryptographic algorithms. Indeed, this trust comes from both a proper security analysis and the attention of external cryptographers, and a document explaining the design choices and trade-offs made is necessary in order to satisfy either of these conditions. We further provide details on how the algorithms allow for optimized implementations (Chapter 5).

1.2 What Are Esch and Schwaemm?

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 5.1 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 5.2.

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 them. The LTS is a strategy which was first used to design the lightweight block cipher Sparx, presented at ASIACRYPT 2016 [DPU+16]. Several independent research teams have already analyzed this algorithm and their results have bolstered our confidence in this design approach. We provide more details about this strategy and third-party results in Section 3.2.

1.2.1 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 1.3). 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 ['ɛʃ].

1.2.2 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 ['ʃvɛm].

1.3 What Is Sparkle?

It is a family of cryptographic permutations based on an ARX design. Its name comes from the block cipher Sparx [DPU+16], 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 1.4.2).

1.4 What Are Their Key Features?

Both SCHWAEMM and ESCH employ the well-known sponge construction. The underlying SPARKLE family of permutation 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 5.

1.4.1 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/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.

1.4.2 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 [DES77], and a linear permutation with a high branching number, like a vast number of SPNs such as the AES [AES01]. 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] [BBCdS+20a]) 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 [BDPVA11, 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 [DEMS16] and the third-round CAESAR candidate KETJE [BDP+16a], both authenticated ciphers, use such an approach. The advantage in this case is a much higher efficiency as we need fewer rounds of the

[1] Alzette is pronounced [alzɛt].

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 [SBK+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.

1.4.3 More Security Features

Security under Random Nonces. All instances of Schwaemm, except 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 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.

[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 $\ll 2^{64}$.

2 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 2.1. Then, we use these permutations to specify the hash functions Esch in Section 2.2 and the authenticated ciphers Schwaemm in Section 2.3.

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

2.1 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 [BBCdS+20a] (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 2.1, 2.2 and 2.3, 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 2.1.1. It is followed by an application of $\mathcal{L}_{n_b}$, a linear permutation operating on all branches; it is specified in Section 2.1.2. 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 2.1. 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 Appendix A. 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 $(m_0, m_1, \ldots, m_{n-1}) \in \mathbb{F}_2^n$, $n \in \{256, 384, 512\}$, is an input to a Sparkle instance, it is mapped to the state words via `state[`k`]` $=$

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

and the inverse mapping is used for transforming state words back to bitstrings.[1]

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

[1] 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.

Definition 2.1.1 (Left/Right branches)**.** *We call* left branches *those that correspond to the state inputs* $(x_0, y_0), (x_1, y_1), \ldots, (x_{n_b/2-1}, y_{n_b/2-1})$*, and we call* right branches *those corresponding to* $(x_{n_b/2}, y_{n_b/2}), \ldots, (x_{n_b-2}, y_{n_b-2}), (x_{n_b-1}, y_{n_b-1})$*.*

Algorithm 2.1 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
  $\big((x_0, y_0), ..., (x_3, y_3)\big) \leftarrow \mathcal{L}_4\big((x_0, y_0), ..., (x_3, y_3)\big)$
end for
return $\big((x_0, y_0), ..., (x_3, y_3)\big)$

Algorithm 2.2 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
  $\big((x_0, y_0), ..., (x_5, y_5)\big) \leftarrow \mathcal{L}_6\big((x_0, y_0), ..., (x_5, y_5)\big)$
end for
return $\big((x_0, y_0), ..., (x_5, y_5)\big)$

Algorithm 2.3 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
  $\big((x_0, y_0), ..., (x_7, y_7)\big) \leftarrow \mathcal{L}_8\big((x_0, y_0), ..., (x_7, y_7)\big)$
end for
return $\big((x_0, y_0), ..., (x_7, y_7)\big)$

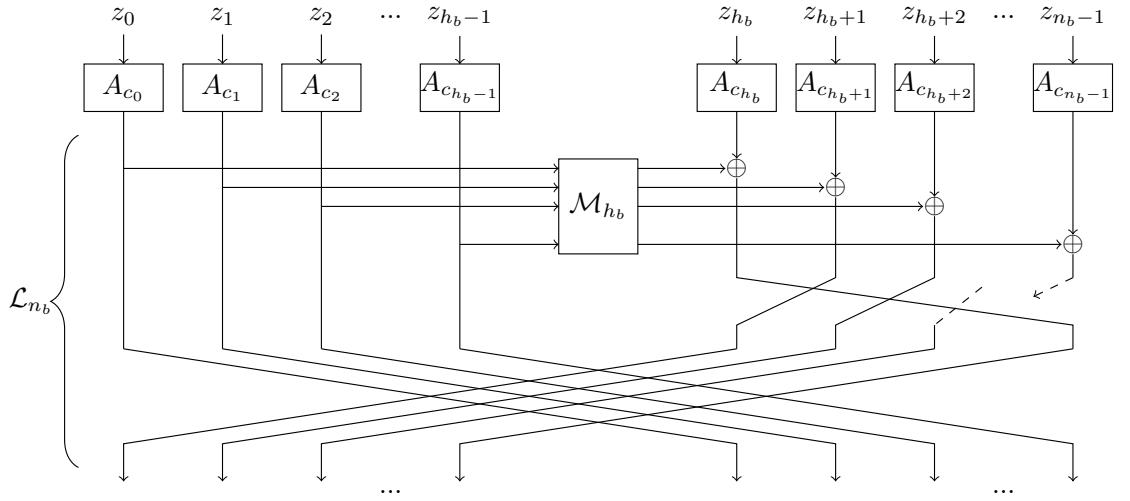


Figure 2.1: The overall structure of a step of SPARKLE. z_i denotes the 64-bit input (x_i, y_i) to the corresponding Alzette instance.

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. Our motivation for this difference is given in Section 3.5. The slim and big versions of all SPARKLE instances are given in Table 2.1.

Table 2.1: The different versions of each SPARKLE instance.

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

2.1.1 The ARX-box Alzette

Alzette, shortly denoted A, is a 64-bit block cipher. It is specified in Algorithm 2.4 and depicted in Figure 2.2. 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 2.1.2). 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. For more details, we provide a complete description of the design choices we made when choosing this component in Section 3.3.

Note. After the round-2 submission, we published the design of Alzette together with its design rationale in [BBCdS+20a]. For completeness, we keep the details of the design process of Alzette in this document.

Algorithm 2.4 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)

2.1.2 The Diffusion Layer

The diffusion layer has a structure which draws heavily from the one used in SPARX-128 [DPU+16]. 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.

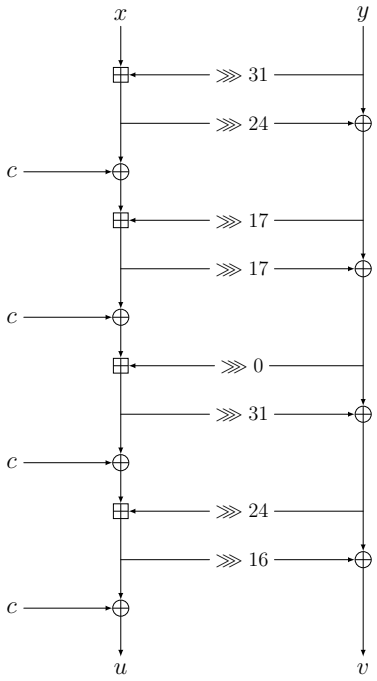


Figure 2.2: The structure of the Alzette instance A_c.

Definition 2.1.2. *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

$$\begin{aligned} t_y &\leftarrow \bigoplus_{i=0}^{w-1} y_i \;,\; t_x \leftarrow \bigoplus_{i=0}^{w-1} x_i \;, \\ u_i &\leftarrow x_i \oplus \ell(t_y),\; \forall i \in \{0, ..., w-1\} \;, \\ v_i &\leftarrow y_i \oplus \ell(t_x),\; \forall i \in \{0, ..., w-1\} \;, \end{aligned} \tag{2.1}$$

where the indices are understood modulo w, and where $\ell : \mathbb{F}_2^{32} \to \mathbb{F}_2^{32}$ 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 2.1. Algorithms describing the three diffusion layers used in our permutations are given in Algorithms 2.5, 2.6 and 2.7. Our rationale for choosing these linear layers is given in Section 3.4.

2.2 The Hash Functions Esch256 and Esch384

2.2.1 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[2] and output a digest D of bitlengths 256, and 384, respectively. Our

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

Algorithm 2.5 $\mathcal{L}_4$
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 $\big((x_0, y_0), \ldots, (x_3, y_3)\big)$

Algorithm 2.6 $\mathcal{L}_6$
Input/Output: $\big((x_0, y_0), \ldots, (x_5, y_5)\big) \in (\mathbb{F}_2^{32} \times \mathbb{F}_2^{32})^6$

▷ Feistel round

$(t_x, t_y) \leftarrow \big(x_0 \oplus x_1 \oplus x_2,\ y_0 \oplus y_1 \oplus y_2\big)$
$(t_x, t_y) \leftarrow \big((t_x \oplus (t_x \ll 16)) \lll 16,\ (t_y \oplus (t_y \ll 16)) \lll 16\big)$
$(y_3, y_4, y_5) \leftarrow (y_3 \oplus y_0 \oplus t_x,\ y_4 \oplus y_1 \oplus t_x,\ y_5 \oplus y_2 \oplus t_x)$
$(x_3, x_4, x_5) \leftarrow (x_3 \oplus x_0 \oplus t_y,\ x_4 \oplus x_1 \oplus t_y,\ x_5 \oplus x_2 \oplus t_y)$

▷ 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 $\big((x_0, y_0), \ldots, (x_5, y_5)\big)$

Algorithm 2.7 $\mathcal{L}_8$
Input/Output: $\big((x_0, y_0), \ldots, (x_7, y_7)\big) \in (\mathbb{F}_2^{32} \times \mathbb{F}_2^{32})^8$

▷ Feistel round

$(t_x, t_y) \leftarrow \big(x_0 \oplus x_1 \oplus x_2 \oplus x_3, y_0 \oplus y_1 \oplus y_2 \oplus y_3\big)$
$(t_x, t_y) \leftarrow \big((t_x \oplus (t_x \ll 16)) \lll 16, (t_y \oplus (t_y \ll 16)) \lll 16\big)$
$(y_4, y_5, y_6, y_7) \leftarrow (y_4 \oplus y_0 \oplus t_x,\ y_5 \oplus y_1 \oplus t_x,\ y_6 \oplus y_2 \oplus t_x,\ y_7 \oplus y_3 \oplus t_x)$
$(x_4, x_5, x_6, x_7) \leftarrow (x_4 \oplus x_0 \oplus t_y,\ x_5 \oplus x_1 \oplus t_y,\ x_6 \oplus x_2 \oplus t_y,\ x_7 \oplus x_3 \oplus t_y)$

▷ 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 $\big((x_0, y_0), \ldots, (x_7, y_7)\big)$

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 2.2 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.

Table 2.2: 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.

	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}

2.2.2 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 2.8 which describes how a block with length strictly smaller that r is turned into a block of length r.

Algorithm 2.8 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

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 Esch256 and Sparkle512$_8$ and Sparkle512$_{12}$ for Esch384. The algorithms are formally specified in Algorithm 2.9 and 2.10 and are depicted in Figure 2.3 and Figure 2.4, 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 $\mathsf{Const_M}$ to the inner part, where $\mathsf{Const_M}$ is different depending on whether the message was padded or not.

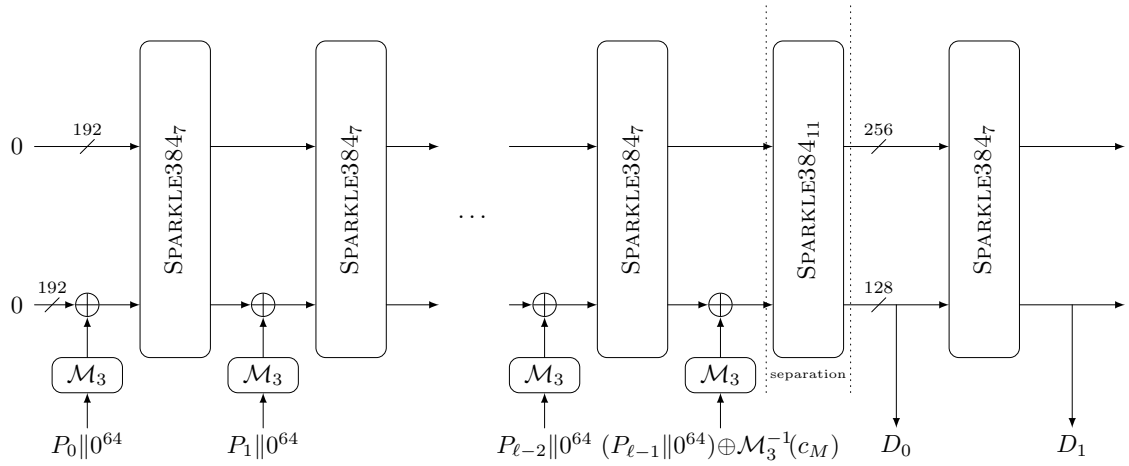


Figure 2.3: The Hash Function Esch256 with rate $r = 128$ and capacity $c = 256$. The constant c_M is equal to $(0, 0, \ldots, 0, 1) \in \mathbb{F}_2^{192}$ if the last block was padded and equal to $(0, 0, \ldots, 0, 1, 0) \in \mathbb{F}_2^{192}$ otherwise.

Algorithm 2.9 ESCH256
Input: $M \in \mathbb{F}_2^*$ *Output:* $D \in \mathbb{F}_2^{256}$

▷ Padding the message
if $M \neq \epsilon$ **then**
  $P_0 \| P_1 \| \ldots \| P_{\ell-1} \leftarrow M$
with $\forall i < \ell - 1$: $|P_i| = 128$ and $1 \leq |P_{\ell-1}| \leq 128$
else
  $\ell \leftarrow 1$
  $P_0 \leftarrow \epsilon$
end if
if $|P_{\ell-1}| < 128$ **then**
  $P_{\ell-1} \leftarrow \mathsf{pad}_{128}(P_{\ell-1})$
  $\mathsf{Const_M} \leftarrow (1 \ll 192)$
else
  $\mathsf{Const_M} \leftarrow (2 \ll 192)$
end if
▷ Absorption
$S \leftarrow 0 \in \mathbb{F}_2^{384}$
for all $j = 0, \ldots, \ell - 2$ **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
$D_0 \leftarrow \mathsf{trunc}_{128}(S)$
$S \leftarrow \text{SPARKLE384}_7\big(S\big)$
$D_1 \leftarrow \mathsf{trunc}_{128}(S)$

return $D_0 \| D_1$

Algorithm 2.10 ESCH384
Input: $M \in \mathbb{F}_2^*$ *Output:* $D \in \mathbb{F}_2^{384}$

▷ Padding the message
if $M \neq \epsilon$ **then**
  $P_0 \| P_1 \| \ldots \| P_{\ell-1} \leftarrow M$
with $\forall i < \ell - 1$: $|P_i| = 128$ and $1 \leq |P_{\ell-1}| \leq 128$
else
  $\ell \leftarrow 1$
  $P_0 \leftarrow \epsilon$
end if
if $|P_{\ell-1}| < 128$ **then**
  $P_{\ell-1} \leftarrow \mathsf{pad}_{128}(P_{\ell-1})$
  $\mathsf{Const_M} \leftarrow (1 \ll 256)$
else
  $\mathsf{Const_M} \leftarrow (2 \ll 256)$
end if
▷ Absorption
$S \leftarrow 0 \in \mathbb{F}_2^{512}$
for all $j = 0, \ldots, \ell - 2$ **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
$D_0 \leftarrow \mathsf{trunc}_{128}(S)$
$S \leftarrow \text{SPARKLE512}_8\big(S\big)$
$D_1 \leftarrow \mathsf{trunc}_{128}(S)$
$S \leftarrow \text{SPARKLE512}_8\big(S\big)$
$D_2 \leftarrow \mathsf{trunc}_{128}(S)$
return $D_0 \| D_1 \| D_2$

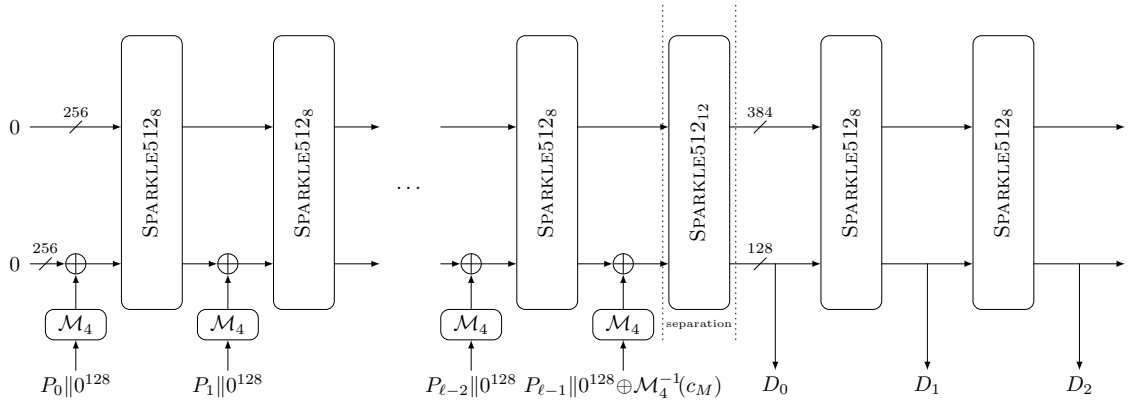


Figure 2.4: The Hash Function ESCH384 with rate $r = 128$ and capacity $c = 384$. The constant c_M is equal to $(0, 0, \ldots, 0, 1) \in \mathbb{F}_2^{256}$ if the last block was padded and equal to $(0, 0, \ldots, 0, 1, 0) \in \mathbb{F}_2^{256}$ otherwise.

2.2.3 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 $\mathsf{Const_M}$ 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 2.11 and 2.12, respectively. The parameters and security levels are given in Table 2.2.

Algorithm 2.11 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**
  $P_0 \| P_1 \| \ldots \| P_{\ell-1} \leftarrow M$
with $\forall i < \ell - 1$: $|P_i| = 128$ and $1 \leq |P_{\ell-1}| \leq 128$
else
  $\ell \leftarrow 1$
  $P_0 \leftarrow \epsilon$
end if
if $|P_{\ell-1}| < 128$ **then**
  $P_{\ell-1} \leftarrow \mathsf{pad}_{128}(P_{\ell-1})$
  $\mathsf{Const_M} \leftarrow (1 \ll 192) \oplus (4 \ll 192)$
else
  $\mathsf{Const_M} \leftarrow (2 \ll 192) \oplus (4 \ll 192)$
end if
▷ Absorption
$S \leftarrow 0 \in \mathbb{F}_2^{384}$
for all $j = 0, \ldots, \ell - 2$ **do**
  $P'_j \leftarrow \mathcal{M}_3(P_j \| 0^{64})$
  $S \leftarrow \textsc{Sparkle384}_7\big(S \oplus (P'_j \| 0^{192})\big)$
end for
$P'_{\ell-1} \leftarrow \mathcal{M}_3(P_{\ell-1} \| 0^{64})$
$S \leftarrow \textsc{Sparkle384}_{11}\big(S \oplus (P'_{\ell-1} \| 0^{192}) \oplus \mathsf{Const_M}\big)$
▷ Squeezing
$D_0 \leftarrow \mathsf{trunc}_{128}(S)$
for all $j = 1, \ldots, \lceil t/128 \rceil - 1$ **do**
  $S \leftarrow \textsc{Sparkle384}_7\big(S\big)$
  $D_j \leftarrow \mathsf{trunc}_{128}(S)$
end for
return $\mathsf{trunc}_t(D_0 \| D_1 \| \ldots \| D_{\lceil t/128 \rceil - 1})$

Algorithm 2.12 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**
  $P_0 \| P_1 \| \ldots \| P_{\ell-1} \leftarrow M$
with $\forall i < \ell - 1$: $|P_i| = 128$ and $1 \leq |P_{\ell-1}| \leq 128$
else
  $\ell \leftarrow 1$
  $P_0 \leftarrow \epsilon$
end if
if $|P_{\ell-1}| < 128$ **then**
  $P_{\ell-1} \leftarrow \mathsf{pad}_{128}(P_{\ell-1})$
  $\mathsf{Const_M} \leftarrow (1 \ll 256) \oplus (4 \ll 256)$
else
  $\mathsf{Const_M} \leftarrow (2 \ll 256) \oplus (4 \ll 256)$
end if
▷ Absorption
$S \leftarrow 0 \in \mathbb{F}_2^{512}$
for all $j = 0, \ldots, \ell - 2$ **do**
  $P'_j \leftarrow \mathcal{M}_4(P_j \| 0^{128})$
  $S \leftarrow \textsc{Sparkle512}_8\big(S \oplus (P'_j \| 0^{256})\big)$
end for
$P'_{\ell-1} \leftarrow \mathcal{M}_4(P_{\ell-1} \| 0^{128})$
$S \leftarrow \textsc{Sparkle512}_{12}\big(S \oplus (P'_{\ell-1} \| 0^{256}) \oplus \mathsf{Const_M}\big)$
▷ Squeezing
$D_0 \leftarrow \mathsf{trunc}_{128}(S)$
for all $j = 1, \ldots, \lceil t/128 \rceil - 1$ **do**
  $S \leftarrow \textsc{Sparkle512}_8\big(S\big)$
  $D_j \leftarrow \mathsf{trunc}_{128}(S)$
end for
return $\mathsf{trunc}_t(D_0 \| D_1 \| \ldots \| D_{\lceil t/128 \rceil - 1})$

2.3 The Authenticated Cipher Family Schwaemm

2.3.1 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[3] 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 [CDNY18], which is based on the well-known SpongeWrap AEAD mode [BDPA11]. The difference between the instances is the version of

[3] As for the hash function, the length can be chosen arbitrarily but it has do be under thresholds that are given in Table 2.3.

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 2.3 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.

Table 2.3: 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.

	n	r	c	$\|K\|$	$\|N\|$	$\|T\|$	security	data limit (in 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}

2.3.2 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 $\mathsf{FeistelSwap}$ 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

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

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 2.5 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 2.13-2.20.

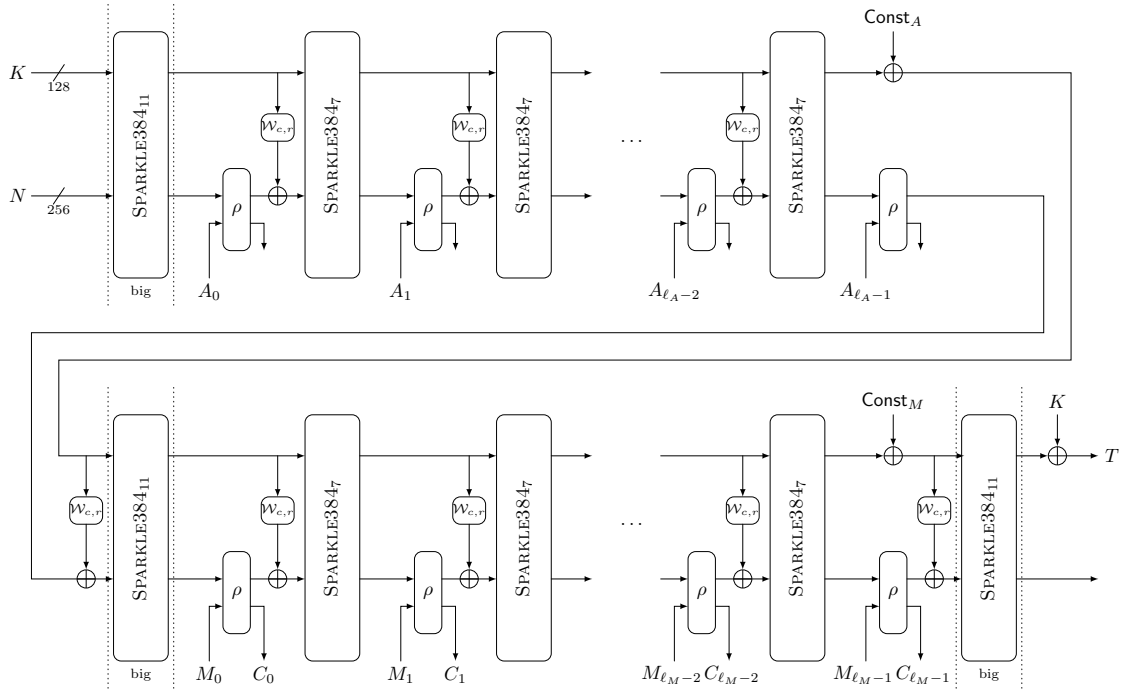


Figure 2.5: The Authenticated Encryption Algorithm SCHWAEMM256-128 with rate $r = 256$ and capacity $c = 128$.

Algorithm 2.13 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**
  $A_0 \| A_1 \| \ldots \| A_{\ell_A - 1} \leftarrow A$ with $\forall i \in \{0, \ldots, \ell_A - 2\} : |A_i| = 256$ 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**
  $M_0 \| M_1 \| \ldots \| M_{\ell_M - 1} \leftarrow M$ with $\forall i \in \{0, \ldots, \ell_M - 2\} : |M_i| = 256$ 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

$S_L \| S_R \leftarrow \textsc{Sparkle384}_{11}\big(N \| K\big)$ with $|S_L| = 256$ and $|S_R| = 128$

▷ Processing of associated data

if $A \neq \epsilon$ **then**
  for all $j = 0, \ldots, \ell_A - 2$ **do**
    $S_L \| S_R \leftarrow \textsc{Sparkle384}_7\big((\rho_1(S_L, A_j) \oplus \mathcal{W}_{128,256}(S_R)) \| S_R\big)$
  end for

▷ Finalization if message is empty

  $S_L \| S_R \leftarrow \textsc{Sparkle384}_{11}\big((\rho_1(S_L, A_{\ell_A - 1}) \oplus \mathcal{W}_{128,256}(S_R \oplus \mathsf{Const}_A)) \| (S_R \oplus \mathsf{Const}_A)\big)$
end if

▷ Encrypting

if $M \neq \epsilon$ **then**
  for all $j = 0, \ldots, \ell_M - 2$ **do**
    $C_j \leftarrow \rho_2(S_L, M_j)$
    $S_L \| S_R \leftarrow \textsc{Sparkle384}_7\big((\rho_1(S_L, M_j) \oplus \mathcal{W}_{128,256}(S_R)) \| S_R\big)$
  end for
  $C_{\ell_M - 1} \leftarrow \mathsf{trunc}_t\big(\rho_2(S_L, M_{\ell_M - 1})\big)$

▷ Finalization

  $S_L \| S_R \leftarrow \textsc{Sparkle384}_{11}\big((\rho_1(S_L, M_{\ell_M - 1}) \oplus \mathcal{W}_{128,256}(S_R \oplus \mathsf{Const}_M)) \| (S_R \oplus \mathsf{Const}_M)\big)$
end if

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

Algorithm 2.14 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**
  $A_0 \| A_1 \| \ldots \| A_{\ell_A - 1} \leftarrow A$ with $\forall i \in \{0, \ldots, \ell_A - 2\} : |A_i| = 256$ 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**
  $C_0 \| C_1 \| \ldots \| C_{\ell_M - 1} \leftarrow C$ with $\forall i \in \{0, \ldots, \ell_M - 2\} : |C_i| = 256$ 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
$S_L \| S_R \leftarrow \textsc{Sparkle384}_{11}\big(N \| K\big)$ with $|S_L| = 256$ and $|S_R| = 128$

▷ Processing of associated data
if $A \neq \epsilon$ **then**
  for all $j = 0, \ldots, \ell_A - 2$ **do**
    $S_L \| S_R \leftarrow \textsc{Sparkle384}_7\big((\rho_1(S_L, A_j) \oplus \mathcal{W}_{128,256}(S_R)) \| S_R\big)$
  end for

▷ Finalization if ciphertext is empty
  $S_L \| S_R \leftarrow \textsc{Sparkle384}_{11}\big((\rho_1(S_L, A_{\ell_A - 1}) \oplus \mathcal{W}_{128,256}(S_R \oplus \mathsf{Const}_A)) \| (S_R \oplus \mathsf{Const}_A)\big)$
end if

▷ Decrypting
if $C \neq \epsilon$ **then**
  for all $j = 0, \ldots, \ell_M - 2$ **do**
    $M_j \leftarrow \rho_2'(S_L, C_j)$
    $S_L \| S_R \leftarrow \textsc{Sparkle384}_7\big((\rho_1'(S_L, C_j) \oplus \mathcal{W}_{128,256}(S_R)) \| S_R\big)$
  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**
    $S_L \| S_R \leftarrow \textsc{Sparkle384}_{11}\big((\rho_1(S_L, \mathsf{pad}_{256}(M_{\ell_M - 1})) \oplus \mathcal{W}_{128,256}(S_R \oplus \mathsf{Const}_M)) \| (S_R \oplus \mathsf{Const}_M)\big)$
  else
    $S_L \| S_R \leftarrow \textsc{Sparkle384}_{11}\big((\rho_1'(S_L, C_{\ell_M - 1}) \oplus \mathcal{W}_{128,256}(S_R \oplus \mathsf{Const}_M)) \| (S_R \oplus \mathsf{Const}_M)\big)$
  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 2.15 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
$S_L \| S_R \leftarrow \textsc{Sparkle384}_{11}\big(N \| K\big)$ with $|S_L| = 192$ and $|S_R| = 192$
▷ Processing of associated data
if $A \neq \epsilon$ **then**
  for all $j = 0, \ldots, \ell_A - 2$ **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 message 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

▷ Encrypting
if $M \neq \epsilon$ **then**
  for all $j = 0, \ldots, \ell_M - 2$ **do**
    $C_j \leftarrow \rho_2(S_L, M_j)$
    $S_L \| S_R \leftarrow \textsc{Sparkle384}_7\big((\rho_1(S_L, M_j) \oplus S_R) \| S_R\big)$
  end for
  $C_{\ell_M - 1} \leftarrow \mathsf{trunc}_t\big(\rho_2(S_L, M_{\ell_M - 1})\big)$
  ▷ Finalization
  $S_L \| S_R \leftarrow \textsc{Sparkle384}_{11}\big((\rho_1(S_L, M_{\ell_M - 1}) \oplus S_R \oplus \mathsf{Const}_M) \| (S_R \oplus \mathsf{Const}_M)\big)$
end if

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

Algorithm 2.16 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 2.17 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**
  $A_0 \| A_1 \| \ldots \| A_{\ell_A - 1} \leftarrow A$ 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 $M \neq \epsilon$ **then**
  $M_0 \| M_1 \| \ldots \| M_{\ell_M - 1} \leftarrow M$ with $\forall i \in \{0, \ldots, \ell_M - 2\} : |M_i| = 128$ 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 $j = 0, \ldots, \ell_A - 2$ **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 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 $j = 0, \ldots, \ell_M - 2$ **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 $(C_0 \| C_1 \| \ldots \| C_{\ell_M - 1}, S_R \oplus K)$

Algorithm 2.18 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**
  $A_0 \| A_1 \| \dots \| A_{\ell_A - 1} \leftarrow A$ 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 $C \neq \epsilon$ **then**
  $C_0 \| C_1 \| \dots \| C_{\ell_M - 1} \leftarrow C$ with $\forall i \in \{0, \dots, \ell_M - 2\} : |C_i| = 128$ 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
$S_L \| S_R \leftarrow \textsc{Sparkle256}_{10}\big(N \| K\big)$ 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 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 $j = 0, \dots, \ell_M - 2$ **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 $(M_0 \| M_1 \| \dots \| M_{\ell_M - 1})$
else
  return $\perp$
end if

Algorithm 2.19 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**
  $A_0 \| A_1 \| \ldots \| A_{\ell_A - 1} \leftarrow A$ with $\forall i \in \{0, \ldots, \ell_A - 2\} : |A_i| = 256$ 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**
  $M_0 \| M_1 \| \ldots \| M_{\ell_M - 1} \leftarrow M$ with $\forall i \in \{0, \ldots, \ell_M - 2\} : |M_i| = 256$ 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 $j = 0, \ldots, \ell_A - 2$ **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 $j = 0, \ldots, \ell_M - 2$ **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 $(C_0 \| C_1 \| \ldots \| C_{\ell_M - 1}, S_R \oplus K)$

Algorithm 2.20 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

2.4 Recommendations for Joint Evaluation

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 2.4. 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.

Table 2.4: Recommendations for joint evaluation of ESCH and SCHWAEMM. The first pairing refers to the primary member of both functionalities.

Hashing	**AEAD**	**based on**
ESCH256	SCHWAEMM256-128	SPARKLE384
ESCH256	SCHWAEMM192-192	SPARKLE384
ESCH384	SCHWAEMM256-256	SPARKLE512
–	SCHWAEMM128-128	SPARKLE256

3 Design Rationale

In this chapter, we explain *why* and *how* we chose the various components of our algorithms. First, we justify the choice of a sponge construction (Section 3.1). Then, we present the motivation behind the overall structure of the permutation in Section 3.2. In particular, we recall the *Long Trail Strategy (LTS)* as it was introduced in the design of SPARX [DPU+16] and explain how it can be adapted to design sponges that are not hermetic but retain very strong security guarantees. Finally, we describe the rationale behind the choice of our two main subcomponents: the ARX-box Alzette (Section 3.3) and the linear layer (Section 3.4). The section will be concluded by a statement on the number of steps used in the permutations (Section 3.5).

Remark on the Notion of a Distinguisher By specifying a fixed cryptographic permutation, i.e., SPARKLE, generic distinguishers that distinguish the permutation from a random one trivially exist. For instance, take the property that the SPARKLE family can be described by a single page of C code, which is not expected to happen for a randomly chosen permutation. When using the term *distinguisher* throughout this document, we actually refer to *structural distinguishers*. In a nutshell, a structural distinguisher allows to obtain information about the internal structure of the permutation or the ability to reverse-engineer it given (a reasonable) number of input-output pairs. Such distinguishers include differential and linear attacks, integral attacks, meet-in-the middle attacks or attacks based on symmetries.

3.1 The Sponge Structure

We decided to use the well-known sponge construction [BDPVA07, BDPVA11] based on a cryptographic permutation. We explain the reasoning behind this decision in this section.

3.1.1 Permutation Versus (Tweakable) Block Cipher

As the aim is to provide authenticated ciphers and hash functions, two broad approaches exist: one based on (tweakable) block ciphers and the sponge structure. Compared to a (tweakable) block cipher, a permutation uses less memory and is conceptually simpler. Because we do not have to evaluate security against related-key distinguishers, we can use the long trail strategy to get bounds for all differential and linear trails that could be leveraged by an attacker. Moreover, as we have a *fixed* permutation, we can conduct statistical experiments. For example we can experimentally check clustering effects for differential or linear trails or the distribution of low-weight monomials (as was for example also done in KECCAK). In a block cipher, every key defines a different permutation and experimentally checking all such permutations would of course be unfeasible.

In summary, we find the sponge to make a better use of memory while allowing the experimental verification of the properties of its building blocks that does not rely on assumptions about the key distribution.

3.1.2 Modes of Operation

We use a sponge mode because of its *provable security* in the idealized setting, i.e., when a random permutation is assumed as its underlying permutation. For hashing, we use the classical sponge mode of operation, similar to that of the NIST standard SHA-3 [Dwo15]. However, we slightly adapt it to allow a minimum-size padding, by employing a similar domain extension scheme as proposed in [Hir16]. In the idealized model, Hirose proved that the corresponding sponge is indifferentiable from a random oracle up to the birthday bound [Hir18].

For authenticated encryption, we use the mode of operation proposed by the designers of BEETLE [CDNY18]. It is a variant of a duplexed sponge [BDPA11]. The reasoning for using this

mode is that it guarantees a security level with regard to confidentiality and integrity (close to) its capacity size in bits instead of an integrity security level of half of the capacity size. It therefore allows us to process more data per permutation call for a given security level and thus to increase the efficiency of our algorithms. We slightly adapted the BEETLE mode by shortening the key to the size of the capacity c, which only adds a term of $\frac{q}{2^c}$ in the bound on the advantage of the adversary (where q denotes the number of permutation queries). We further shortened the tag to the size of the capacity to limit the increase in the ciphertext size and adapted the handling in case of empty associated data and message. We further XOR the key before outputting the tag. We finally changed the particular constants $\mathsf{Const_A}$ and $\mathsf{Const_M}$ for domain extension by encoding the capacity size into them. This differentiates the SCHWAEMM instances that use the same underlying SPARKLE permutations.

Our use of big and slim permutations means that the security of our high level algorithms is not directly implied by standard provable security arguments. However, a careful analysis of various cryptanalysis techniques allows us to claim with a high confidence that this discrepancy is not important in our case.

On Sponge-based Constructions in the Past. The *sponge* construction for hashing was introduced at the ECRYPT hash workshop in 2007 ([BDPVA07]) and it was proven that a random sponge is indistinguishable from a random oracle up to inner collisions, thus imposing a security level of $\frac{c}{2}$ bits, where c denotes the capacity. At EUROCRYPT 2008, it was shown that the sponge is also indifferentiable of a random oracle up to complexity $2^{\frac{c}{2}}$ in the ideal permutation (resp., ideal function) model [BDPV08]. In this setting, the adversary has also access to the random permutation (resp., function) implemented in the sponge. The sponge as an underlying construction was adopted in the NIST standard SHA-3 [Dwo15] in 2015. We point to [BDPVA11] for a detailed documentation of the sponge and its variants, including duplexed sponges. They were introduced at SAC 2011 [BDPA11] and allow to both inject and output messages between each permutation call. When initialized with a secret key, the authors showed that a duplexed sponge can simply be applied for authenticated encryption in a mode called SPONGEWRAP, for which they proved a security level of min $\left(\kappa, \frac{c}{2}\right)$ bits in the ideal permutation model, where κ denotes the key size.

The elegant construction of a duplexed sponge inspired lots of cryptographers to submit authenticated ciphers to the CAESAR competition that were based on such a construction, e.g., the winner for the use case of lightweight applications ASCON [DEMS16], the third-round candidates KETJE [BDP+16a], KEYAK [BDP+16b] and NORX [AJN16], and many more first and second-round competitors.

In [JLM14], the authors derived improved security bounds on sponge-based constructions for authenticated encryption (i.e., duplexed sponge constructions). Their main result was an improved security level on confidentiality of min $\left(\frac{b}{2}, c, \kappa\right)$ bits, where b denotes the block size of the permutation, c the capacity, and κ the size of the key. The authors derived a similar improved security for integrity under the assumption that the number of forgery attempts of the adversary is restricted to fulfill the inequality $q_p + \sigma_E + \sigma_D \leq \frac{2^c}{\sigma_D}$, where q_p denotes the number of permutation queries, σ_E the number of encryption queries and σ_D the number of forgery attempts (see [JLM14, Theorem 2]). This improved result affected many CAESAR competitors and some teams reduced the capacity size in their submissions (e.g., NORX and ASCON-128a). However, in their Journal of Cryptology version of the paper [JLM+18], the authors explicitly mention that "caution must be taken" if the ciphers with reduced capacity are used in applications in which the forgery attempts by the adversary are not limited. They refer to a discussion on that topic on the CAESAR mailing list.[1]

In [ADMA15], the authors considered the keyed duplex sponge construction more generally (i.e., not only in the use case of authenticated encryption) and proved its indistinguishability from a random oracle, also allowing the adversary access to the underlying permutation, up to a security level of c. Thus they obtain a result similar to [JLM14], but, unlike in the latter paper, the authors of [ADMA15] do not restrict themselves to specific authenticated ciphers. Moreover, they analyze the indistinguishability in multi-key settings.

[1] https://groups.google.com/forum/#!topic/crypto-competitions/YZ3sMMzzXro

In [CDNY18], the authors proposed a variation of the duplexed sponge mode, called Beetle, that uses a combined feedback function ρ for injecting and producing message/ciphertext blocks. While in classical duplexed sponges the ciphertext blocks equal the outer part of the internal states, the feedback function does not leak information about the internal state from a ciphertext only. The authors proved a security of their construction of $\min\left(r, \frac{b}{2}, c - \log_2(r)\right)$, both for confidentiality and integrity, *and without the restriction on the number of forgery attempts.* Here, r denotes the size of the rate in bits.

3.1.3 Improving Sponge-based Modes

As our approach is not hermetic, our choices are guided by the best attack that can be found against the permutations *in a mode*. In order to mitigate some of them, we propose some simple modifications to the sponge-based modes we use. These changes are equivalent to alterations of the permutation used, meaning that they are compatible with the sponge structure.

3.1.3.1 Rate Whitening

In a sponge-based authenticated cipher, the security of the primitive is based on the secrecy of the inner part. Hence, we can safely allow the adversary to read the content of the outer part. However, in practice, this can allow the attacker to compute a part of the permutation. Indeed, if the outer part is aligned with the S-box layer then the attacker can evaluate said S-box layer on the outer part. In our case, as half of the linear layer is the identity function, it would allow the attacker to partially evaluate two steps of the permutation. It is not clear what advantage they could derive from such observations as the content of the inner part remains secret in this case. However, it is easy to prevent this phenomena using what we call *rate whitening*. It simply consists in XORing branches from the inner part into the outer part just before the permutation call. That way, the attacker cannot evaluate a part of the permutation without first guessing parts of the inner part.

This modification to the mode can be instead interpreted as the use of an altered permutation which contains the rate whitening. Thus, this improvement to a sponge-based mode is compatible with said mode.

3.1.3.2 Indirect Injection

In a sponge-based hash function, an r-bit message block is XORed into the outer part. In Esch, it is not exactly the case. Instead, the r-bit message block is first expanded into a larger message using a linear function and the result is injected into the state of the sponge. We call this pre-processing of the message blocks "indirect injection".

As with rate whitening, the purpose of this modification is to alleviate potential issues arising when the outer part is aligned with the S-box layer. Indeed, in such a case, the attacker does not need to find a differential trail covering the whole permutation to find a collision. Instead, they can find a differential covering all but the first and last layers of S-boxes which will propagate through this layer with probability 1.

In order to prevent such attacks, it is sufficient to modify the injection procedure so that the space in which the injected message lies is not aligned with the S-box layer. To this end, we reuse the linear Feistel function used in our Sparkle instances. For example, in Esch256, we do not inject message branches x and y directly but, instead, inject the 3-branch message $\mathcal{M}_3(x, y, 0)$. This is equivalent to using a regular injection while composing the permutation with an application of $\mathcal{M}_3$ in the input and one of $\mathcal{M}_3^{-1}$ in the output, so that this modification still yields a "regular sponge".

This simple modification to the injection procedure efficiently disrupts the alignment between the outer part and the Alzette layer. Furthermore, because the linear functions we use for the indirect injection ($\mathcal{M}_3$ and $\mathcal{M}_4$) have a branching number of 4, and because only two branches are injected through them, we know that at least two double Alzette instances are activated during message injection. Similarly, a differential trail yielding a possible cancellation by an indirectly injected message in the output of a permutation implies that two double Alzette instances are active in the end of the trail. Because this pattern cannot be truncated, it means that a differential trail

mapping an indirectly injected difference to an indirectly injected difference has a probability upper-bounded by the double ARX-box bound to the power 4. As the best differential trail covering a double ARX-box has a probability at most equal to 2^{-32}, we deduce the following lemma.[2]

Lemma 3.1.1. *The probability (taken over all inputs) of a differential trail that is introduced and then cancelled via indirectly injected messages after at least one iteration of* Sparkle384 *or* Sparkle512 *is at most equal to* 2^{-128}.

The general principle consisting in applying a linear code to the message block before injection is reminiscent of the technique used to input the message blocks in the SHA-3 candidate Hamsi [Küç09].

In our case, messages that have a length multiple of r are not padded, instead, a constant is added into the state of the sponge that is outside the control of the adversary. This constant is added on the left part of the state to ensure its diffusion but, at first glance, we might think that it could be cancelled via a difference in a message block since the indirect injection XORs data into the whole left part of the state. However, since the constant is only over a single word, a difference cancelling it would have to span 3 (respectively 4) input branches of the linear permutation used for indirect injection in Esch256 (resp. Esch384). Because we fix 1 (resp. 2) inputs of this linear permutation to 0, a direct application of Theorem 3.4.1 shows that no message difference can cancel this constant.

3.2 A Permutation Structure that Favours Rigorous Security Arguments

After settling on the design of a permutation, we need to decide how to build it. The structure used must allow strong arguments to be made for the security it offers against various attacks while being amenable to very efficient implementations in terms of code size, RAM usage and speed. First, we present the mathematical framework of provable security against differential and linear attacks (Section 3.2.1). Then we present the *Long Trail Strategy (LTS)* as introduced in the design of Sparx[3] (Section 3.2.2). Finally, we argue that the use of the LTS allows us to bound the probability of all differential/linear trails, including those that are obtained by absorbing (possibly many) blocks into a sponge (Section 3.2.3). Thus, it allows us to have some guarantees even if the permutation "in a vacuum" has some distinguishers. In other words, it allows us to build non-hermetic algorithms with the same security arguments as hermetic ones.

3.2.1 Provable Security Against Differential and Linear Attacks

The resistance of a symmetric-key primitive against differential and linear cryptanalysis is determined by the differential (resp. linear) characteristic/s with maximum probability (resp. absolute correlation). The reason is that the success probability of a differential (resp. linear) attack depends on the amount of data (number of plaintexts) necessary to execute the attack. The latter is, in turn, proportional to the inverse of the probability (resp. absolute correlation) of the best differential (resp. linear) characteristic.

For keyed constructions the maximum N-round probability (resp. absolute correlation) for a fixed key is approximated by the expected maximum probability (resp. absolute correlation) over all keys. This is known as assuming the *Hypothesis of Stochastic Equivalence* (see e.g. [LMM91] [DR02, § 8.7.2, pp. 121]).

We denote the two quantities – the maximum expected differential trail (or characteristic) probability and the maximum expected absolute linear trail (or characteristic) correlation – respectively by MEDCP and MELCC. These abbreviations have been previously used in the literature e.g. in [KS07].

For computing the MEDCP and MELCC we work under the assumption of independent round keys. The latter allows us to compute the probability of an N-round characteristic as the product

[2] In the specific cases of the Esch functions, an attacker could try and leverage the padding scheme and the different constants added in the outer part to add a difference to the state in a way which is not coherent with indirect injections. Still, our LTS-derived differential bounds (see Section 4.3.1) allow us to simply solve this problem.

[3] As hinted by its name, Sparkle is a descendent of the block cipher Sparx. In fact, this block cipher was co-designed by members of our team.

of its corresponding 1-round transitions. This is also known as assuming the *Hypothesis of Independent Round Keys* (see e.g. [DR02, § 8.7.2, pp. 121]). Note that, since we are in the permutation setting, we do not have any round keys. Therefore, this assumption indeed doesn't hold technically. However, we have validated experimentally that it is a good approximation for what happens in practice.

We prove that the proposed designs – Schwaemm and Esch – are resistant against differential and linear attacks by showing that for the underlying permutation Sparkle, there does not exist differential and linear characteristics with MEDCP and MELCC that are high enough to be exploited in an attack. The tools that make it possible to prove such statements lie in the heart of the *Long Trail Strategy*, which will be presented in more detail next.

3.2.2 The Long Trail Strategy

The *Long Trail Strategy (LTS)* is a design approach that was introduced by the designers of Sparx [DPU+16] to bound the differential probabilities and absolute linear correlations for ARX-based primitives with large internal states.

Up to that point, the only formal bounds available for ARX-based algorithms were obtained via computer search which, for computational reasons, were restricted to small block sizes (mostly 32 bits, possibly up to 64) [BVC16]. The LTS is an approach that allows the construction of round functions operating on a much larger state in such a way that the bounds obtained computationally over a small state can be used to derive bounds for the larger structure. This very high level description is virtually identical to that of the *Wide Trail Strategy (WTS)*, introduced in [Dae95] and famously used to design the AES [AES01]. However, the specifics of these two methods are very different (see Section 3.2.2.3). First, we recall how a long trail argument works to bound the differential probabilities and absolute linear correlations (Section 3.2.2.1).

3.2.2.1 The Long Trail Argument

In what follows, we focus on the case of differential probabilities. The linear case is virtually identical. In order to build a cipher according to the LTS, we need:

- a *non-linear operation* A operating on b bits such that the differential probability for multiple iterations of A is bounded,
- a *linear layer* operating on b-bit branches.

The bound is then computed by looping over all the truncated trails that are allowed by the linear layer. As the number of b-bit branches is low (in our case, at most 8) and as the linear layer is sparse (in our case, half of the outputs are copies of the input), this loop is very efficient. Then, for each truncated trail, we perform two operations.

1. First, we decompose the truncated into *long trails*. A long trail is a continuous differential trail at the branch level that receives no difference from other words. If r iterations of A are performed on a branch without any call to the linear layer, then the probability of all differential trails that fit in this truncated trail is at most equal to the bound for r rounds of A. More subtly, if $x \leftarrow A^r\left(A^r(x) \oplus L(y)\right)$ and the difference over y is equal to 0 then we can bound the differential probability by the one corresponding to $2r$ rounds of A.

   The decomposition of a truncated trail into its constitutive *long trails* is obtained by grouping all the chains of t active branches that do not receive differences from the outside into *long trails* of length t.

2. In order to bound the probability of all differential trails that fit in a truncated trail over r rounds, we use

$$\prod_{t=1}^{r} p_t \times n_t \ ,$$

   where p_t is the bound for t rounds of A and where n_t is the number of long trails of length t in the truncated trail.

Example 3.2.1. *Here, we reproduce the example given in the specification of* SPARX *[DPU+16].*

Consider a 64-bit block cipher using a 32-bit S-box, one round of Feistel network as its linear layer and 4 steps without a final linear layer. Consider the differential trail $(\delta_0^L, \delta_0^R) \to (\delta_1^L, \delta_1^R) \to (0, \delta_2^R) \to (\delta_3^L, 0)$ *(see Fig. 3.1 where the zero difference is dashed). Then this differential trail can be decomposed into 3 long trails represented in black, blue and red: the first one has length 1 and* δ_0^R *as its input; the second one has length 2 and* δ_0^L *as its input; and the third one has length 3 and* δ_1^L *as its input so that the long trail decomposition of this trail is* $\{t_1 = 1, t_2 = 1, t_3 = 1\}$*, where* t_i *denotes the number of of long trails of length* i*.*

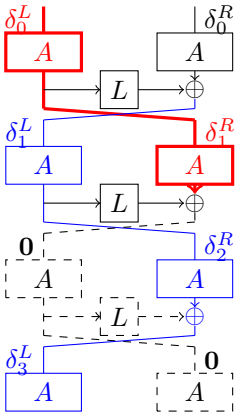


Figure 3.1: The decomposition into long trails of a truncated trail in a simple cipher.

A good structure to leverage long trail is the one described in Figure 3.2. By forcing the chaining of multiple rounds of A in each branch and in each step, it ensures the existence of some long trails. In order to further exploit the long trails, we can set L to be essentially a Feistel round defined by

$$(x_0, ..., x_{i-1}), (y_0, ..., y_{i-1}) \mapsto \big(y_0 \oplus \ell_0(x_0, ..., x_{i-1}), ..., y_{i-1} \oplus \ell_{i-1}(x_0, ..., x_{i-1})\big), (x_0, ..., x_{i-1}) \ ,$$

where the ℓ_i are linear functions operating on i branches. Indeed, such linear trails ensure the existence of long trails of length $2r$ because half of the inputs are copied to the output. At the same time, the diffusion provided by a Feistel round is well understood and it is the same in both the forward and backward directions.[4]

These observations led us to use such a linear layer when designing SPARX. Now that SPARX has undergone third-party cryptanalysis (see Section 3.2.2.2) we confidently reuse this structure.

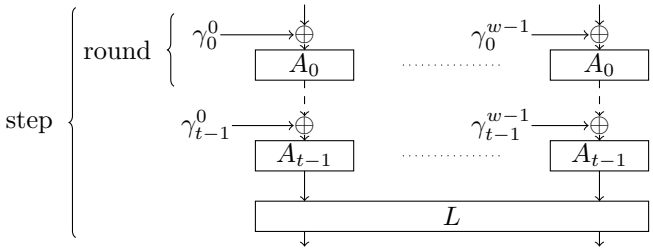


Figure 3.2: The overall structure of a step for the long trail strategy. The wires correspond to *branches* which, in our case, are divided into two *words*.

3.2.2.2 Literature on SPARX and the LTS

The sanity of the long trail strategy was confirmed by several cryptanalysis attempts targetting SPARX. In [AL18], the authors describe some attacks against round-reduced versions of the ciphers

[4] Again, we stress that we assume independent calls to Alzette and use bounds on the MEDCP/MELCC, although we don't have round keys.

and confirm that no differential trail with a probability higher than the one found via the long trail argument exists. In a follow-up work presented at SAC'18 [AK18], the authors found that the differential effect is not a threat to the security of Sparx, i.e. that the clustering of differential trails was rather small—as in all ARX-based ciphers. Other authors have analyzed Sparx without finding any significant attack [ATY17, TAY17].

The LTS served as the basic principle for the design of other algorithms by another team. The sLiSCP permutation [ARH+17] was designed with hardware efficiency in mind. It uses a generalized Feistel structure with a few rounds of the block cipher Simeck [YZS+15] as the Feistel function. The bounds for the differential probability and absolute linear correlations were obtained using a long trail argument. It was later improved by replacing the generalized Feistel structure with a generalized MISTY structure, thus saving some copies of the state. Again, long trail arguments were used by the designers. The result, sLiSCP-light, was published later [ARH+18].

3.2.2.3 LTS versus WTS

The wide trail strategy (WTS), famously used to design the AES, is the most common design strategy for block ciphers and permutations. Thus, we provide a quick comparison of these two approaches.

Bound Derivation. For the WTS, the diffusion must ensure a high number of active S-boxes in differential and linear trails. The bound on the corresponding primitive is derived using p^a where p is the relevant probability at the S-box level and a is the number of active S-boxes. The aim is then to increase a. In contrast, in the LTS, the bound is derived by looping over all possible truncated trails, decomposing each into its long trails and computing the bound accordingly.

Confusion. In algorithms built using the WTS, the non-linearity is provided by *S-boxes*, small functions operating on typically 4 or 8 bits. In contrast, in the LTS, the non-linearity is provided by multiple rounds of a more complex function operating on a much larger state (32 bits in the case of Sparx, 64 bits for Sparkle).

Diffusion. The diffusion layer in the WTS must ensure a high number of active S-boxes. In the LTS, it is more subtle: if a difference does not propagate, it might prevent the interruption of a long trail which could counter intuitively lead to a lower probability than if the difference did propagate. Nevertheless, in order for the cipher to resist other attacks, the diffusion layer must provide some diffusion. We have found that Feistel-based linear layers provided a good compromise between these two requirements.

Two-Staged Security Analysis. At the heart of both design strategies lies the idea of separating the analysis of the cipher into two stages. First, we study the non-linear part (be it its small S-box or its wide ARX-box) and then, using properties of the linear layer, we deduce the properties of the cipher. This two stage approach simplifies the task of the cryptanalyst as it allows the use of computer assisted method to investigate the properties of the non-linear part (which operate on a small enough block size that it is possible, i.e. at most 64 bits for an ARX-box). Hence, ciphers designed with either the WTS or the LTS are easier to study than more classical ARX designs.

3.2.3 Applying the LTS to Absorption

In a sponge function, the state is divided into two parts: the *outer part* is r-bit long and the *inner part* is c-bit long. The quantities r and c are respectively called the *rate* and the *capacity*. Regardless of the use of the sponge, r-bit plaintext blocks are XORed into the outer part of the sponge.

3.2.3.1 Hermetic versus Not-Hermetic Approach

When building a block cipher, designers ensure that their algorithm is safe from differential and linear attacks. The methods to prove resilience against these attacks are well known and they help provide a good estimate of the number of rounds needed to ensure security against these attacks.

For sponge-based hash functions and authenticated ciphers, the best current approach for provable security consists in building a permutation such that no distinguisher exist for it. This security will then naturally carry over to the hash function. However, this approach provides protection against an unrealistic adversary: in a sponge function, the adversary can only control the outer part of the internal state. Therefore, we can expect that preventing all such distinguishers is over-engineering, especially in a lightweight setting. Unfortunately, it is difficult to estimate the number of rounds needed in the permutation to prevent attacks from more realistic adversaries, i.e. ones which can only modify the content of the outer part.

Like for block ciphers, we turn to the prevention of differential and linear attacks to make this estimation. Differential attacks pose a serious threat to hash functions as evidenced by the practical attack against SHA-1 [SBK+17]; and linear biases in the keystream generated by several authenticated ciphers have been identified as well, for instance in MORUS [AEL+18]. In Sections 3.2.3.2 and 3.2.3.3 respectively, we show that dangerous respectively differential and linear distinguishers can be proven to have a negligible probability when the algorithm considered is a sponge with a permutation built using the LTS.

3.2.3.2 Preventing Differential Attacks in Sponges

Differential trails that could be useful for an adversary trying to attack a sponge are prevented in two stages.

First, for hashing, we use a permutation call between the absorption phase and the squeezing phase that does not yield any differential with probability higher than 2^{-s}, where s is the security parameter. More formally, we want the permutation which is called between absorption and squeezing to have the following property.

Property 3.2.1 (Absorption/Squeezing Separation)**.** *Let $P : \mathbb{F}_2^r \times \mathbb{F}_2^c \to \mathbb{F}_2^r \times \mathbb{F}_2^c$ be a permutation. It* separates the two phases *with a security level of s bits if:*

$$\forall \delta \in \mathbb{F}_2^r \times \mathbb{F}_2^c, \delta \neq 0 \implies \forall \Delta \in \mathbb{F}_2^r \times \mathbb{F}_2^c, \Pr[P(x \oplus \delta) \oplus P(x) = \Delta] \leq 2^{-s} \ ,$$

where the probability is taken over all $x \in \mathbb{F}_2^r \times \mathbb{F}_2^c$.

This property is essentially what we would expect of a random permutation except that the bound on the probability is 2^{-s} rather than 2^{-c-r}. The point in this case is to destroy any pattern that could exist in the internal state of the sponge before squeezing, even if this pattern is only in the inner part. In other words, Property 3.2.1 ensures that no non-trivial differential pattern can be exploited once the squeezing phase has been reached.

We then need to ensure that no differential trail ending with an all-zero difference (a collision) exists with a probability higher than 2^{-s}. If we can show this absence then we expect that the probability of existence of a valid pair with specific predefined input values is upper-bounded by 2^{r-s}.

Property 3.2.2 (Unfeasibility of Vanishing Differences)**.** *Let $P : \mathbb{F}_2^r \times \mathbb{F}_2^c \to \mathbb{F}_2^r \times \mathbb{F}_2^c$ be a permutation and let P_m be the permutation of $\mathbb{F}_2^r \times \mathbb{F}_2^c$ parameterized by $m \in \mathbb{F}_2^r$ defined by*

$$P[m] : (x, y) \mapsto P(x \oplus m, y) \ .$$

Furthermore, for any integer $a > 0$ and fixed differences $\delta_0, \ldots, \delta_{a-1} \in \mathbb{F}_2^r$, let $\mathsf{P}_{\mathsf{vanish}}(a)$ be the probability that

$$\big(P[x_{a-1} \oplus \delta_{a-1}] \circ \cdots \circ P[x_0 \oplus \delta_0]\big)(y) \ \oplus \ \big(P[x_{a-1}] \circ \cdots \circ P[x_0]\big)(y) \ = \ (0, 0)$$

where the probability is taken over all $(x_0, ..., x_{a-1}) \in (\mathbb{F}_2^r)^a$ and all $y \in \mathbb{F}_2^r \times \mathbb{F}_2^c$.

We say that P makes vanishing differences unfeasible *for a security parameter s if, for all numbers of absorbed message blocks a, we have*

$$\forall (\delta_0, ..., \delta_{a-1}) \in (\mathbb{F}_2^r)^a, \delta_0 \neq 0 \implies \mathsf{P}_{\mathsf{vanish}}(a) \leq 2^{-s} \ .$$

This property is different from the absorption/squeezing separation. Indeed, we are not looking at any differential trail but specifically at those that correspond to the absorption of r-bit blocks. Similarly, we only worry about differences that end up only in the outer part of the sponge as these are the differences that can be cancelled via the absorption of a message. For hashing, we are aiming for a security parameter of $r + \frac{c}{2}$. As already explained in the documentation of cryptographic sponges [BDPVA11, Section 8.4.1.1], if the probability of an outer part to outer part differential trail can be upper-bounded by $2^{-r-\frac{c}{2}}$, the expected number of pairs following the trail is upper-bounded by $2^{\frac{c}{2}}$. Thus, the probability that there exist a valid pair with a fixed and predefined input *value* is $\leq 2^{-\frac{c}{2}}$.

Hypothesis 3.2.1. *If a permutation of $\mathbb{F}_2^r \times \mathbb{F}_2^c$ satisfies both the absorption/squeezing separation and the unfeasability of vanishing differences with security parameter $r + c/2$ then it can be used to construct a sponge for which a differential collision search cannot be more efficient than a basic collision search.*

The aim of such a differential attack would be to find a pair of messages x, y such that either of the following two happens:

1. after absorbing x and y, the states of the sponges are identical (collision of the full state)
2. there is a difference between the two states but, after squeezing, this difference is not over a part of the state that matters.

If the first case has a probability higher than $2^{-r-\frac{c}{2}}$ then the unfeasibility of vanishing differences is violated. If the second one is, then the separation of squeezing and absorption is violated. Hence, satisfying both prevents such attacks.

Because of the uniqueness of the nonce, attacks on the decryption oracle (e.g., differential forgeries) in AEAD are the most dangerous kind of attack that exploit differential cryptanalysis. For AEAD, we therefore aim for a security level of c when considering outer part to outer part trails.

In practice, designers usually cannot prove that those properties are satisfied without any simplifying assumptions. Still, we can assume that the probability of differential trails is a good approximation for the probability of the differentials and then we can prove, using a variant of the long trail argument, that no differential *trail* can falsify either property. The conclusion is then that the sponge is safe from a differential collision search.

We have observed that those two requirements require different number of steps in the iterated permutation. It makes sense as the attacker has full control over the difference in the absorption/squeezing separation case while it can only inject some specific differences in the vanishing differences case. The hermetic sponge strategy [BDPVA11, Section 8.1.1] then consists of the case where $s = r + c$. While the hermetic sponge strategy certainly yields secure sponges, our finer approach allows us to use fewer steps. In particular, our approach can be expected to yield different number of steps during absorption and between absorption and squeezing.

In order to bound the probability of differential trails that are relevant for Property 3.2.1, we can simply consider the permutation like a block cipher and use the probability bounding techniques that are relevant given its design strategy. In our case, we simply reuse the long trail argument that was introduced in SPARX, i.e. we loop over all truncated trails, divide them into long trails and deduce a probability bound for each. This method can be efficiently implemented using a variant of Matsui's search, as explained in Section 4.2.2.1.

For Property 3.2.2, using an LTS-based permutation simplifies the search greatly. Usually, the search space corresponding to the search for the trails considered in Property 3.2.2 is too large. Indeed, in this case, the differences are injected in the outer part of the sponge *during each absorption*. We therefore multiply the set of possible input differences by 2^r each time we consider an absorption. It means that Property 3.2.2 is a priori impossible to verify unless we simply ensure that Property 3.2.1 holds for the same number of steps.

However, for the SPARKLE permutation family, this finer search is possible.

As we bound the differential probabilities using the LTS, we first enumerate all possible truncated differential trails and then bound the probabilities for each individual truncated trail. As the branches are wide (64-bit in our case), each message injections lead to the addition of $r/64$

bits of information in terms of truncated trails. It is thus possible to enumerate all truncated trails covering a certain amount of message absorption where the difference is injected only in the outer part. The details of the algorithm we used in the case of SPARX are given in Section 4.3.1.

3.2.3.3 Preventing Linear Attacks in Sponges

Mirroring differentials, we can define linear approximations that are of particular interest for cryptanalysts and whose absolute correlation we must strive to lower. The main such correlations corresponds to the following property.

Property 3.2.3 (Undetectability of Keystream Bias)**.** *Let $P : \mathbb{F}_2^r \times \mathbb{F}_2^c \to \mathbb{F}_2^r \times \mathbb{F}_2^c$ be a permutation. We say it has* undetectable keystream biases *with security parameter s if the absolute correlation of each linear approximation of P^i involving only bits in the outer part is lower than $2^{-s/2}$, for all numbers of iterations i where i is smaller than the order of the permutation.*

Hypothesis 3.2.2. *Let $P : \mathbb{F}_2^r \times \mathbb{F}_2^c \to \mathbb{F}_2^r \times \mathbb{F}_2^c$ be a permutation that has undetectable keystream biases with security parameter s. If it is used to construct a sponge-based stream cipher then it is impossible to distinguish its output from a random stream using linear biases.*

Detection of biases is a dangerous attack on AEAD schemes. We aim for a security parameter of $s = d$, where d equals the binary logarithm of the number of blocks allowed by the data limit.

As for the differential case, we do not know how to prove that all linear approximation have such a low absolute correlation. However, we can approximate the absolute correlations of linear approximations by those of the linear trails yielding them, and then upperbound the absolute correlations of said trails.

3.3 The ARX-box Alzette

In this section, we present both the design process and the main properties of Alzette as published in [BBCdS+20a]. A summary of those properties is provided in Section 3.3.11.

3.3.1 Round Structure and Number of Rounds

We decided to build Alzette out of the operations *XOR of rotation* and *ADD of rotation*, i.e., $x \oplus (y \ggg s)$ and $x + (y \ggg r)$, because they can be executed in a single clock cycle on ARM processors and thus provide extremely good diffusion per cycle. As the Alzette instances will be implemented with their rounds unrolled, we allowed the use of different rotations in every round. We observed that one can obtain much better resistance against differential and linear attacks in this case compared to having identical rounds.

In particular, we aimed for designing an ARX-box of the form depicted in Figure 3.3, where each word is of size 32 bits and which iterates t rounds. The i-th round is defined by the rotation amounts $(r_i, s_i) \in \mathbb{Z}_{32} \times \mathbb{Z}_{32}$ and the round constant $(\gamma_i^L, \gamma_i^R) \in \mathbb{F}_2^{32} \times \mathbb{F}_2^{32}$.

In our final design of Sparkle, we decided to use $t = 4$ rounds. The reason is that, for r-round ARX-boxes, usable LTS bounds can be obtained from the $2r$-round bounds of the ARX structure by concatenating two ARX-boxes (see Section 3.2.2.1). The complexity of deriving upper bounds on the differential trail probability or absolute linear trail correlation depends on the number of rounds considered. For 8 rounds, i.e., 2 times a 4-round ARX-box, it is feasible to compute strong bounds in reasonable time (i.e., several days up to few weeks on a single CPU). For 3-round ARX-boxes, the 6-round bounds of the best ARX-boxes we found seem not strong enough to build a secure cipher with a small number of steps. Since we cannot arbitrarily reduce the number of steps in Sparkle because of structural attacks, using ARX-boxes with more than four rounds would lead to worse efficiency overall. In other words, we think that four-round ARX-boxes provide the best balance between the number of steps and rounds per step in order to build a secure permutation.

3.3.2 Criteria for Choosing the Rotation Amounts

We aimed for choosing the rotations (r_i, s_i) in Alzette in a way that maximizes security and efficiency. For efficiency reasons, we want to minimize the *cost* of the rotations, where we use the

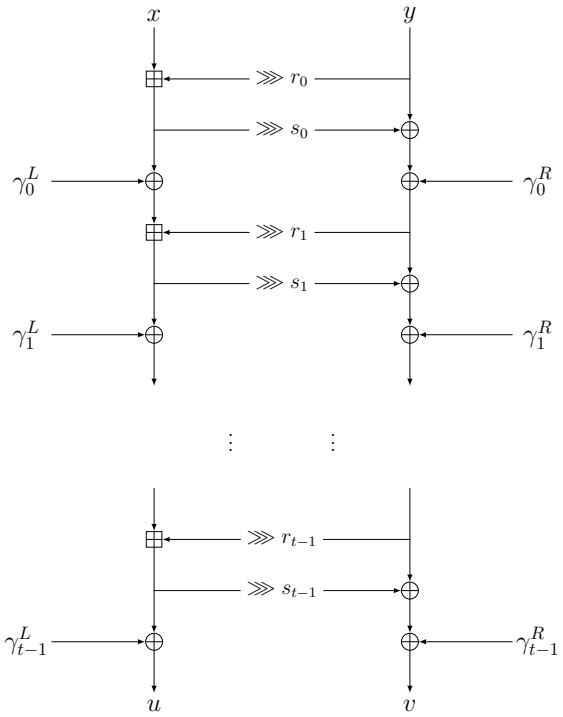


Figure 3.3: The general structure of Alzette.

cost metric as given in Table 3.1. While each rotation has the same cost in 32-bit ARM processors, we further aimed for minimizing the cost with regard to 8-bit and 16-bit architectures. Therefore, we restricted ourselves to rotations from the set $\{0, 1, 7, 8, 9, 15, 16, 17, 23, 24, 25, 31\}$, as those are the most efficient when implemented on 8 and 16-bit microcontrollers. We define the *cost* of a collection of rotation amounts (that is needed to define all the rounds of an ARX-box) as the sum of the costs of its contained rotations.

Table 3.1: For each rotation in $\{0, 1, 7, 8, 9, 15, 16, 17, 23, 24, 25, 31\}$, the table shows an estimation of the number of clock cycles needed to implement the rotation on top of XOR, resp. ADD. We associate the mean of those values for the three platforms to be the *cost* of a rotation.

rot (mod 32)	8-bit AVR	16-bit MSP	32-bit ARM	cost
0	0	0	0	0.00
± 1	5	3	0	2.66
± 7	5	9	0	4.66
8	0	6	0	2.00
± 9	5	9	0	4.66
± 15	5	3	0	2.66
16	0	0	0	0.00

For security reasons, we aim to minimize the provable upper bound on the expected differential trail probability (resp. expected absolute linear trail correlation) of a differential (resp. linear) trail. More precisely, our target was to obtain strong bounds, preferably at least as good as those of the round structure of the 64-bit block cipher Speck, i.e., an 8-round differential bound of 2^{-29} and an 8-round linear bound of 2^{-17}. If possible, we aimed for improving upon those bounds. Note that for $r > 4$, the term *r-round bound* refers to the differential (resp. linear) bound for r rounds of an iterated ARX-box. As explained above, at the same time we aimed for choosing an ARX-box with a low cost. In order to reduce the search space, we relied on the following criteria as a heuristic for selecting the final choice for Alzette:

- The candidate ARX-box must fulfill the differential bounds ($-\log_2$) of 0, 1, 2, 6, and 10 for 1, 2, 3, 4 and 5 rounds respectively, for *all four possible offsets*. We conjecture that those bounds are optimal for up to 5 rounds.
- The candidate must fulfill a differential bound of at least 16 for 6 rounds, also for all offsets.
- The 8-round linear bound ($-\log_2$) of the candidate ARX-box should be at least 17.

By the term *offset* we refer to the round index of the starting round of a differential trail. Note that we are considering all offsets for the differential criteria because the bounds are computed using Matsui's branch and bound algorithm, which needs to use the $r-1$-round bound of the differential trail with starting round index 2 in order to compute the r-round bound of the trail.

We tested *all* rotation sets with a cost below 12 for the above conditions. None of those fulfilled the above criteria. For a cost below 15, we found the ARX-box with the rotations as presented in Table 3.2. The first two lines correspond to the choice for Alzette.

Table 3.2: Differential and linear bounds for our choice of rotation parameters with all four offsets. For each offset, the first line shows $-\log_2 p$, where p is the maximum expected differential trail probability for the differential case and the second line shows $-\log_2 c$, where c is the maximum expected absolute linear trail correlation for the linear case. The value set in parenthesis corresponds to the maximum absolute correlation of the linear hull taking clustering into account, derived by the experimental verification. The bounds [BVC16, FWG+16, LWR16, Liu17, LLJW21] for SPECK are given at the bottom of the table for comparison.

$(r_0, r_1, r_2, r_3, s_0, s_1, s_2, s_3)$	1	2	3	4	5	6	7	8	9	10	11	12
$(31, 17, 0, 24, 24, 17, 31, 16)$	0	1	2	6	10	18	26	≥ 32	≥ 36	≥ 42	≥ 46	≥ 52
	0	0	1	2	5	8	13 (11.64)	17 (15.79)	–	–	–	–
$(17, 0, 24, 31, 17, 31, 16, 24)$	0	1	2	6	10	17	25	31	≥ 37	≥ 41	≥ 47	–
	0	0	1	2	5	9	13	16	–	–	–	–
$(0, 24, 31, 17, 31, 16, 24, 17)$	0	1	2	6	10	18	25	≥ 32	≥ 36	≥ 42	–	–
	0	0	1	2	6	8	13	15	–	–	–	–
$(24, 31, 17, 0, 16, 24, 17, 31)$	0	1	2	6	10	17	25	≥ 31	≥ 37	–	–	–
	0	0	1	2	5	9	12	16	–	–	–	–
SPECK64	0	1	3	6	10	15	21	29	34	38	42	46
	0	0	1	3	6	9	13	17	19	21	24	27

3.3.3 On the Differential Properties of Alzette

We used Algorithm 1 of [BVC16], adapted to our round structure, to compute the bounds on the maximum expected differential trail probabilities of the ARX-boxes. This algorithm is basically a refined variant of Matsui's well-known branch and bound algorithm [Mat95]. While the latter has been originally proposed for ciphers that have S-boxes (in particular – DES), the former is targeted at ARX-based designs that use modular addition, rather than an S-box, as a source of non-linearity.

Algorithm 1 [BVC16] exploits the differential properties of modular addition to efficiently search for characteristics in a bitwise manner. Upon termination, it outputs a trail (characteristic) with the maximum expected differential trail probability (MEDCP). For Alzette, using search as in [Liu17, LLJW21] we could obtain such trails for up to seven rounds, where the 7-round bound is 2^{-26}. We further collected all trails corresponding to the maximum expected differential probability for 4 and 5 rounds and experimentally checked the actual probabilities of the differentials, see Section 3.3.10.1.

Note that for 8 rounds, we could not get tight bounds due to the high complexity of the search. In other words, the algorithm didn't terminate in reasonable time. However, the algorithm exhaustively searched the range up to $-\log_2(p) = 31$ for 8 rounds, which proves that there are no valid differential trails with an expected differential trail probability greater than 2^{-32}.

3.3.4 On the Linear Properties of Alzette

We used the Mixed-Integer Linear Programming approach described in [FWG+16] and the Boolean satisfiability problem (SAT) approach in [LWR16] in order to get bounds on the maximum expected absolute linear trail correlation. It was feasible to get tight bounds even for 8 rounds, where the 8-round bound for Alzette is 2^{-17}. We were able to collect all linear trails that correspond to the maximum expected absolute linear trail correlation for 4 up to to 8 rounds and experimentally checked the actual correlations of the corresponding linear approximations, see Section 3.3.10.2.

3.3.5 Diffusion in Alzette

Alzette provides very fast diffusion. In particular, all outputs bits depend on all the input bits after 4 rounds, though this dependency can be very weak. After 8 rounds however, we have that all output bits strongly depend on all the input bits. This strong diffusion ensures that three steps of the SPARKLE permutations already fulfill the strict avalanche criterion, see Section 4.2.1.

3.3.6 Invariant Subspaces

Invariant subspace attacks were considered in [LAAZ11]. Using a similar to the "to and fro" method from [PGC98, BDBP03], we searched for an affine subspace that is mapped by an ARX-box A_{c_i} to a (possibly different) affine subspace of the same dimension. We could not find any such subspace of nontrivial dimension.

Note that the search is randomized so it does not result in a proof. As an evidence of the correctness of the algorithm, we found many such subspace trails for all 2-round reduced ARX-boxes, with dimensions from 56 up to 63. For example, let A denote the first two rounds of A_{c_0}. Then for all $l, r, l', r' \in \mathbb{F}_2^{32}$ such that $A(l, r) = (l', r')$,

$$(l_{29} + r_{21} + r_{30})(l_{30} + r_{31})(l_{31} + r_0)(r_{22})(r_{23}) = \\ (l'_4 + r'_{21})(l'_5 + r'_{22})(l'_6 + r'_{23})(l'_{28} + l'_{30} + l'_{31} + r'_{13} + 1)(l'_{29} + l'_{31} + r'_{14}).$$

This equation defines a subspace trail of constant dimension 59.

3.3.7 Nonlinear Invariants for Alzette

Nonlinear invariant attacks were considered recently in [TLS16]. Using linear algebra, we experimentally verified that for any Alzette instance A_{c_i} and any non-constant Boolean function f of degree at most 2, the compositions $f \circ A_{c_i}$ and $f \circ A_{c_i}^{-1}$ have degree at least 10:

$$\forall f \colon \mathbb{F}_2^{64} \to \mathbb{F}_2, 1 \le \deg(f) \le 2 \quad \deg(f \circ A_{c_i}) \ge 10, \deg(f \circ A_{c_i}^{-1}) \ge 10,$$

and for functions f of degree at most 3, the compositions have degree at least 4:

$$\forall f \colon \mathbb{F}_2^{64} \to \mathbb{F}_2, 1 \le \deg(f) \le 3 \quad \deg(f \circ A_{c_i}) \ge 4, \deg(f \circ A_{c_i}^{-1}) \ge 4.$$

In particular, any A_{c_i} has no cubic invariants. Indeed, a cubic invariant f would imply that $f \circ A_{c_i} + \varepsilon = f$ is cubic (for a constant $\varepsilon \in \mathbb{F}_2$). The same holds for the inverse of any Alzette instance A_{c_i}.

By using the same method, we also verified that there are no quadratic equations relating inputs and outputs of any A_{c_i}. However, there are quadratic equations relating inputs and outputs of 3-round reduced versions of each A_{c_i}.

3.3.8 Linearization of Alzette instances

In recent attack against KECCAK instances [QSLG17, SLG17] the S-box linearization technique is used. The idea is to find a subset of inputs (often an affine subspace), such that the S-box acts linearly on this set. We attempted to linearize Alzette instances by finding all inputs for which all four modular additions inflict no carry bits and thus are equivalent to XOR. For the addition of two random independent 32-bit words, the probability of having all carry bits equal to zero is

equal to $(3/4)^{31}$. Indeed, for each bit position, if no carry comes in, then the outgoing carry will occur only if both input bits are equal to 1. Furthermore, the carry bit from the most significant bits is ignored. Assuming independence of the additions in Alzette, $2^{64}/(3/4)^{124} \approx 2^{12.5}$ inputs are expected to satisfy the linearization.

In order to find all inputs, we have to solve a system of quadratic equations. Indeed, for the first round, the condition is $(x \,\&\, (y \ggg 31)) \lll 1 = 0$ (left shift by one omits the most significant bit), which provides 31 quadratic bit equations. Since this condition ensures that the output of the first round is linear, we get similar quadratic equations for the second round, except that x and y are replaced with corresponding linear functions. In total we obtain 124 quadratic equations of the form $l(x, y) \cdot r(x, y) = 0$, where $l, r \colon \mathbb{F}_2^{64} \to \mathbb{F}_2$ are affine. We solved this system by a guess-and-determine method with a few optimizations, for all Alzette constants that we use. The results are given in Table 3.3.

constant	hexadecimal	# inputs	example
c_0	`b7e15162`	13	(`05600000`, `70000225`)
c_1	`bf715880`	11	(`2a001990`, `00188000`)
c_2	`38b4da56`	18	(`1000c000`, `144a0528`)
c_3	`324e7738`	3	(`1000e620`, `04270080`)
c_4	`bb1185eb`	10	(`001c8181`, `10808201`)
c_5	`4f7c7b57`	340	(`08301013`, `28265722`)
c_6	`cfbfa1c8`	105	(`801d8000`, `2fd10085`)
c_7	`c2b3293d`	76	(`00220110`, `20001804`)
0	`00000000`	8	(`00000000`, `40200080`)
$2^{32} - 1$	`ffffffff`	$\geq 2^{22}$	(`0b11cc51`, `72770942`)

Table 3.3: The number of inputs for ARX-boxes inflicting no carries in all four rounds, for different round constants.

The first interesting observation is that the number of solutions is much smaller than $2^{12.5} \approx 5900$ predicted under the round independence assumption. For 5 out of 8 used constants, the number of solutions is less than 20, and the maximum number of solutions is 340. The second observation is that, for the zero constant, the number of solutions is also extremely low. We find it rather counter-intuitive, since in absence of constants many low-weight vectors can be expected to pass through Alzette without inflicting any carries. On the other hand, it turns out that the all-one constant leads to a larger than expected number of solutions. We observed a similar behaviour and verified the correctness of our algorithm on 8-bit words, where we performed an exhaustive search over all Alzette inputs.

We suggest that main reason behind the small numbers of solutions is the strong ARX-box structure itself, in particular the rotation amounts we used. Note however that other linearization methods are possible, for example by fixing particular non-zero carry patterns.

3.3.9 On the Round Constants

The purpose of round constant additions, i.e., the XORs with γ_i^L, γ_i^R in the general ARX-box structure, is to ensure some independence between the rounds. Furthermore, and in fact much more importantly, they should ensure that the Alzette instances called on each branch are independent: If we use the same round constants in all the branches then the permutation is very weak because of existing symmetries. The fact that such weak constant choice exist is not a bad thing in and on itself, it simply means that we must use round constants that are different enough in each branch.

For efficiency reasons, we decided to use the same round constant in every round of the ARX-box, i.e., $\forall i : \gamma_i^L = c$. Moreover, we chose all $\gamma_i^R = 0$. It is important to note that the experimental verification of the differential probabilities and absolute linear correlations we conducted did not lead to significant differences when changing to a more complex round constant schedule. In other words, even for random choices of all γ_i^L and γ_i^R, we did not observe significantly different results that would justify the use of a more complex constant schedule (which would of course lead to worse efficiency in the implementation).

In order to be transparent in the way we selected the constants, we derived the eight different 32-bit constants $c_0, \ldots, c_7$ for the eight Alzette instances from the fractional digits of $e = 2.71\ldots$. In particular, we converted the number into its base-16 representation and choose c_0 to be the first block of eight fractional digits, c_1 as the third block, c_2 as the 6th, c_3 as the 9th, c_4 as the 14th, c_5 as the 15th, c_6 as the 26th and c_7 as the 29th block. We excluded several blocks in order to leverage some observed linear hull effects in our experimental verification for 5 and 6 rounds to our favor. We give more details on that in Section 3.3.10 below.

3.3.10 Experimental Verifications

3.3.10.1 Experiments on the Fixed-Key Differential Probabilities

As in virtually all block cipher designs, the security arguments against differential attacks are only average results when *averaging over all keys of the primitive*. When leveraging such arguments for a cryptographic permutation, i.e., a block cipher with a fixed key, it might be possible in theory that the actual fixed-key maximum differential probability is higher than the expected maximum differential probability. In particular, the variance of the distribution of the maximum fixed-key differential probabilities might be high.

For all of the 8 Alzette instances used in Sparkle (depending on the constant c_i), we conducted experiments in order to see if the expected maximum differential trail probabilities derived by Matsui's search are close to the actual differential probabilities of the fixed instances. Our results are as follows.

By Matsui's search we found 7 differential trails for the four-round ARX-box[5] that correspond to the maximum expected differential trail probability of 2^{-6}, see Table 3.4. For any Alzette instance A_{c_i} and any such trails with input difference α and output difference β, we experimentally computed the actual differential probability of the differential $\alpha \to \beta$ by

$$\frac{|\{x \in S | A_{c_i}(x) \oplus A_{c_i}(x \oplus \alpha) = \beta\}|}{|S|},$$

where S is a set of 2^{24} inputs sampled uniformly at random. Our results show that the expected differential trail probabilities approximate the actual differential probabilities very well, i.e., all of the probabilities computed experimentally are in the range $[2^{-6} - 10^{-4}, 2^{-6} + 10^{-4}]$ for a sample size of 2^{24}.

For 5 rounds, i.e., one iteration of Alzette and one additional first round of Alzette, there is only one trail with maximum expected differential trail probability $p = 2^{-10}$. For all *combinations* of round constants that can occur in 5 rounds (one iteration of Alzette plus one round) that do not go into the addition of a step counter, i.e., corresponding to the twelve ARX-box compositions

$$\begin{array}{llllll} A_{c_2} \circ A_{c_0} & A_{c_3} \circ A_{c_1} & A_{c_3} \circ A_{c_0} & A_{c_4} \circ A_{c_1} & A_{c_5} \circ A_{c_2} & A_{c_4} \circ A_{c_0} \\ A_{c_5} \circ A_{c_1} & A_{c_6} \circ A_{c_2} & A_{c_7} \circ A_{c_3} & A_{c_2} \circ A_{c_3} & A_{c_3} \circ A_{c_4} & A_{c_2} \circ A_{c_7}, \end{array}$$

we checked whether the actual differential probabilities are close to the maximum expected differential trail probability. We found that all of the so computed probabilities are in the range $[2^{-10} - 10^{-5}, 2^{-10} + 10^{-5}]$ for a sample size of 2^{28}.

3.3.10.2 Experiments on the Fixed-Key Linear Correlations

Similarly as for the case of differentials, for all of the 8 Alzette instances used in Sparkle, we conducted experiments in order to see whether the maximum expected absolute linear trail correlations derived by MILP and presented in Table 3.2 are close to the actual absolute correlations of the linear approximations over the fixed instances. Our results are as follows, and presented in Table B.1 in Appendix B.

For four rounds, there are 4 trails with a maximum expected absolute trail correlation of 2^{-2}. For all of the eight Alzette instances, the actual absolute correlations are very close to the theoretical

[5]Note that those are independent of the actual round constants as the probability corresponds to the average probability over all keys when analyzing Alzette as a block cipher where independent subkeys are used instead of round constants.

Table 3.4: The input and output differences α, β (in hex) of all differential trails over Alzette corresponding to maximum expected differential trail probability $p = 2^{-6}$ and $p = 2^{-10}$ for four and five rounds, respectively.

rounds	α	β	$-\log_2(p)$
4	8000010000000080	8040410041004041	6
	8000010000000080	80c04100410040c1	6
	0080400180400000	8000018081808001	6
	0080400180400000	8000008080808001	6
	a0008140000040a0	8000010001008001	6
	8002010000010080	0101000000030101	6
	8002010000010080	0301000000030301	6
5	a0008140000040a0	8201010200018283	10

values and we did not observe any clustering. For more than four rounds, we again checked all combinations of Alzette instances that do not get a step counter. For five rounds, there are 16 trails with a maximum expected absolute trail correlation of 2^{-5}. In our experiments, we can observe a slight clustering. The observed absolute correlations based on 2^{24} samples can also be found in Table B.1. The minimum and maximum refers to the minimum, resp., maximum observed absolute correlations over all the combinations of Alzette instances that do not get a step counter, similar as tested for differentials. We chose the initial round constants c_i such that, for all combinations of ARX-boxes that occur over the linear layer, the linear hull effect is to our favor, i.e., the actual absolute correlation tends to be *lower* than the theoretical value. For the excluded blocks of the fractional digits of e, the actual absolute correlations are slightly higher (but all smaller than 2^{-8}).

This tendency also holds for the correlations over six rounds. There are 48 trails with a maximum expected absolute linear trail correlation of 2^{-8}. The results of our experiments for 2^{28} random samples are shown in Table B.2 in Appendix B.

For seven rounds, there are 2992 trails with a maximum expected absolute linear trail correlation of 2^{-13}. Over all the twelve combinations that don't add a step counter and all of the 2992 approximations, the maximum absolute correlation we observed was $2^{-11.64}$ using a sample size of 2^{32} plaintexts chosen uniformly at random.

For eight rounds, there are 3892 trails with a maximum expected absolute linear trail correlation of 2^{-17}. Over all the twelve combinations that don't add a step counter and all of the 3892 approximations, the maximum absolute correlation we observed was $2^{-15.79}$ using a sample size of 2^{40} plaintexts chosen uniformly at random.

3.3.10.3 Experimental Algebraic Degree Lower Bound

The modular addition is the only non-linear operation in Alzette. Its algebraic degree is 31 and thus, in each 4-round ARX-box, there must exist some output bits of algebraic degree at least 32.

We experimentally checked that, for each A_{c_i}, the algebraic degree of *each* output bit is at least 32. In particular, for each output bit we found a monomial of degree 32 that occurs in its ANF. Note that for checking whether the monomial $\prod_{i=0}^{m-1} x_{i_m}$ occurs in the ANF of a Boolean function f one has to evaluate f on 2^m inputs.

3.3.10.4 Division Property of the ARX-box Structure

Division property is a technique introduced by Todo [Tod15] to find *integral characteristics*. Originally, it was applied to substitution-permutation networks and Feistel networks. Later, *bit-based division property* was proposed by Todo and Morii [TM16] and applied to the Simon block cipher with 32-bit blocks. Due to the high computation complexity of the search algorithm, it is infeasible to apply the technique to ciphers with larger block sizes. However, Xiang *et al.* [XZBL16] discovered that the bit-based division property propagation can be efficiently encoded as *an mixed-integer linear programming* instance (MILP), and, surprisingly, can be solved on practice using modern optimization software (Gurobi Optimizer [GO18]) for practically all known block ciphers. Sun *et*

Table 3.5: Look-up table of f.

input	output	input	output
000	00	100	01
001	01	101	10
010	01	110	10
011	10	111	11

Table 3.6: Division property propagation table of f.

input	outputs	input	outputs
000	{00}	100	{01, 10}
001	{01, 10}	101	{10}
010	{01, 10}	110	{10}
011	{10}	111	{11}

al. [SWW17] described a way to encode the modular addition operation using MILP inequalities, extending the framework to ARX-based primitives.

For further information on division property propagation and its encoding using MILP inequalities, we refer to [XZBL16]. However, we describe briefly a new technique for encoding division property propagation through the modular addition. Our technique is simpler and more compact than the one proposed by Sun *et al.* [SWW17].

Addition modulo 2^{32}. The method by Sun *et al.* is based on expressing the modular addition as a Boolean circuit and applying the standard known encoding for XORand ANDoperations. As a result, for each bit of a word at least 12 bit operations are produced. We propose a new simple method which requires only 2 *inequalities* per bit.

Our key idea is to compute the carry bits and the output bits in pairs using a 3×2 bit look-up table. The division property propagation through this look-up table can be encoded using only 2 inequalities.

Consider an addition of two n-bit words $a, b \in \mathbb{F}_2^n$ and let $y = a \boxplus b \mod 2^n$ (recall that a_0 denotes the most significant bit of a, a_{n-1} denotes the least significant bit of a, etc.). Define *carry* bits $c_i, -1 \leq i < n$ as follows: $c_{n-1} = 0$ and $c_i = \mathrm{Maj}(a_{i+1}, b_{i+1}, c_{i+1})$ for $-1 \leq i < n-1$, where Maj is the 3-bit *majority* function. Then it is easy to verify that $y_i = a_i \oplus b_i \oplus c_i$ for all $0 \leq i < n$. Full modular addition can be computed sequentially from $i = n-1$ to $i = 0$. Let $f : \mathbb{F}_2^3 \to \mathbb{F}_2^2$ be such that $f(a, b, c) = (\mathrm{Maj}(a, b, c), a \oplus b \oplus c)$, then we can write

$$(c_{i-1}, y_i) = f(a_i, b_i, c_i),$$

for all $0 \leq i < n$. The lookup table of f is given in Table 3.5. Note that no bits are copied in the sequential computation process. It follows that the division property propagation can be encoded directly by encoding n sequential applications of f (using the S-Box encoding methods by Xiang *et al.* [XZBL16]). Finally, an additional constraint is needed to ensure that the resulting division property is not active in the bit c_{-1}.

The division property propagation table is given in Chapter 3.6. This table can be characterized by the two following integer inequalities:

$$\begin{cases} -a - b - c + 2c' + y & \geq 0, \\ a + b + c - 2c' - 2y & \geq -1, \end{cases}$$

where $a, b, c \in \mathbb{Z}_2$ correspond to the values of the input division property and $c', y \in \mathbb{Z}_2$ correspond to the values of the output division property. In our experiments, these two inequalities applied for each bit position generate precisely the correct division property propagation table of the addition modulo 2^n for n up to 7. There are a few redundant transitions, but they do not affect the result.

An alternative to MILP-solvers that is used for division property analysis are SMT-solvers. To facilitate this alternative method, we characterize the division property propagation table of f by four Boolean propositions (obtained by enumerating all possible outputs and constraining respective inputs):

$$\begin{cases} c' \wedge y & \Rightarrow & a \wedge b \wedge c, & \triangleright\ a = b = c = 1 \\ \neg c' \wedge \neg y & \Rightarrow & \neg a \wedge \neg b \wedge \neg c, & \triangleright\ a = b = c = 0 \\ \neg c' \wedge y & \Rightarrow & (a \oplus b \oplus c) \wedge (\neg a \vee \neg b), & \triangleright\ a + b + c = 1 \\ c' \wedge \neg y & \Rightarrow & (a \vee b \vee c) \wedge (\neg a \vee \neg b \vee \neg c). & \triangleright\ 1 \leq a + b + c \leq 2 \end{cases}$$

We used this representation together with the Boolector SMT-solver [NPB15] (version 3.1.0) to verify our results.

For completeness, we also provide a pure SAT encoding (CNF - a conjunction normal form) of the propagation through f (11 clauses):

$$(a \vee b \vee c \vee \neg y) \wedge (a \vee b \vee c \vee \neg c') \wedge (a \vee \neg c' \vee \neg y) \wedge (b \vee \neg c' \vee \neg y)$$
$$\wedge (\neg c \vee c' \vee y) \wedge (\neg b \vee c' \vee y) \wedge (\neg b \vee \neg c \vee c') \wedge (\neg a \vee c' \vee y)$$
$$\wedge (\neg a \vee \neg c \vee c') \wedge (\neg a \vee \neg b \vee c \vee \neg y) \wedge (\neg a \vee \neg b \vee \neg c \vee y).$$

Finally, we note that subtraction modulo 2^n, used in the inverse of Alzette, is equivalent to the addition with respect to the division property propagation in our method. Indeed, let $f' : \mathbb{F}_2^3 \to \mathbb{F}_2^2$,

$$f'(a, b, c) = (c', y) = ([a - b - c < 0], a \oplus b \oplus c),$$

where the first coordinate of f' computes the subtraction carry bit. It is in fact equivalent to the first coordinate of f (the majority function) up to XORwith constants:

$$\begin{aligned}[a - b - c < 0] &= [a + (1 - b) + (1 - c) < 2] \\ &= 1 - [a + (1 - b) + (1 - c) \geq 2] = 1 \oplus f_0(a, 1 \oplus b, 1 \oplus c).\end{aligned}$$

We conclude that f' has the same division property propagation table as f and thus division property propagation using our method is the same for modular addition and subtraction.

Division property propagation in Alzette. First, we evaluated the general algebraic degree of the ARX-box structure based on the division property. The 5^{th} and 6^{th} rounds rotation constants were chosen as the 1^{st} and 2^{nd} rounds rotation constants respectively, as this will happen when two Alzette instances will be chained. The inverse ARX-box structure starts with 4^{th} round rotation constants, then 3^{rd}, 2^{nd}, 1^{st}, 4^{th}, etc. The minimum and maximum degree among coordinates of the ARX-box structure and its inverse are given in Table 3.7. Even though these are just upper

Table 3.7: The upper bounds on the minimum and maximum degree of the coordinates of Alzette and its inverse.

ARX-box rounds	1	2	3	4	ARX-box inverse rounds	1	2	3	4
min	1	10	42	63	min	1	2	32	46
max	32	62	63	63	max	32	62	63	63

bounds, we expect that they are close to the actual values, as the division property was shown to be rather precise [TM16]. Thus, the ARX-box structure may have full degree in all its coordinates, but the inverse of the ARX-box has a coordinate of degree 46.

The block-size level division property of Alzette is such that, for any $1 \leq k \leq 62$, $\mathcal{D}_k^{64}$ maps to $\mathcal{D}_1^{64}$ after two rounds, and $\mathcal{D}_{63}^{64}$ maps to $\mathcal{D}_2^{64}$ after two rounds and to $\mathcal{D}_1^{64}$ after three rounds. The same holds for the inverse structure of Alzette.

The longest integral characteristic found with bit-based division property is for 6-round ARX-box, where the input has 63 active bits and the inactive bit is at the index 44 (i.e., there are 44 active bits from the left and 19 active bits from the right), and in the output 16 bits are balanced:

input active bits:

`11111111111111111111111111111111,11111111111101111111111111111111`,

balanced bits after 6-round ARX-box:

`??????????????????????????BBBBBBBB,??????????BBBBBBBB??????????????`.

The inactive bit can be moved to indexes $45, 46, 47, 48$ as well, the balanced property after 6 round stays the same. For the 7-round ARX-box we did not find any integral distinguishers.

For the inverse ARX-box, the longest integral characteristic is for 5 rounds:

input active bits:
`11111111111111111111111101111,11111111111111111111111111111111`,
balanced bits after 5-round ARX-box inverse:
`??????????????????????????????B,???????BBBBBBBB??????????????????`.

For the ARX-box inverse with 6-rounds we did not find any integral characteristic.

As a conclusion, even though a single Alzette iteration has integral characteristics, for two chained Alzette instances there are no integral characteristics that can be found using the state-of-the-art division property method.

3.3.11 Summary of the Properties of Alzette

Our experimental results validate our theoretical analysis of the properties of Alzette: in practice, the differential and linear trail probabilities (resp., absolute correlations) are as predicted. In the case of differential probabilities, the clustering is minimal. While it is not quite negligible in the linear case, it does not pose a threat to Sparkle in practice (see Section 4.2.3).

The diffusion is fast: all output bits depend on all input bits after a single call of Alzette– though the dependency may be sometimes weak. After a double Alzette call, diffusion is of course complete. Furthermore, as evidenced by our analysis of the division property of our component, no integral distinguisher exist in this case.

While the two components have utterly different structures, our ARX-box Alzette has similar properties to one round of AES and the double ARX-box to its super-S-box (see Table 3.8). The bounds for the (double) iteration of Alzette come from Table 3.2. For the AES, the bounds for a single rounds are derived from the properties of its S-box, so its maximum differential probability is $4/256 = 2^{-6}$ and its maximum absolute linear correlation is 2^{-3}. For two rounds, the differential trail bound is 2^{-30} and the linear one is 2^{-15}: as the branching number of the MixColumn operation is 5, we raise the quantities of the S-box to the power 5.

Table 3.8: A comparison of the properties of Alzette with those of the AES with a fixed key. MEDCP denotes the maximum expected differential trail probability and MELCC denotes the maximum expected absolute linear trail correlation.

	MEDCP	MELCC
Alzette	2^{-6}	2^{-2}
AES S-box layer	2^{-6}	2^{-3}
Double Alzette	$\leq 2^{-32}$	2^{-17}
AES super S-box layer	2^{-30}	2^{-15}

We emphasize that our experimental verifications were enabled by our use of a key-less structure. For a block cipher, we would need to look at all possible keys to reach the same level of confidence.

3.4 The Linear Layer

3.4.1 Overall Structure

The Alzette layer ensures that no pattern exists at the branch level that a cryptanalyst could leverage to mount an attack. However, we need to provide diffusion between the branches. Furthermore, as the Alzette layer needs two steps (i.e., 8 rounds) to obtain good differential and linear bounds, we have to follow the long trail approach and ensure that long trails exist in all differential and linear trails. Hence, our linear layer has to provide two apparently opposite properties: diffusion between the branches and little diffusion to help fostering long trails.

To solve this problem, we copy the technique that we initially introduced when designing SPARX and use a linear layer with a Feistel structure. Intuitively, it leaves one half of the state unchanged and thus ensures the existence of long trails. At the same time, the Feistel function itself provides excellent diffusion, meaning that we can quickly ensure that all branches in the state depend on all the branches in the input.

3.4.2 The Linear Feistel Function

In what follows, we establish several lemmas and theorems that describe the behaviour of the linear Feistel functions $\mathcal{M}_w$ that are used in SPARKLE instances.

Lemma 3.4.1. *Let $w > 2$ be an integer. If w is even then the inverse of $\mathcal{M}_w$ is computed as*

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

i.e., it is $\mathcal{M}_w$ itself. On the other hand, if w is odd, it is computed as

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

Proof. The proof in the even case is very straight-forward because $\bigoplus_{i=0}^{w-1} x_i = \bigoplus_{i=0}^{w-1} u_i$ and $\bigoplus_{i=0}^{w-1} y_i = \bigoplus_{i=0}^{w-1} v_i$. Let us therefore consider the case where w is odd.

In order to obtain (x_i, y_i) from (u_i, v_i), we need to obtain the values of $\ell(t_x)$ and $\ell(t_y)$ from the (u_i, v_i). We remark that

$$\begin{aligned} t_u &= \bigoplus_{i=0}^{w-1} (x_i \oplus \ell(t_y)) = t_x \oplus \ell(t_y), \\ t_v &= \bigoplus_{i=0}^{w-1} (y_i \oplus \ell(t_x)) = t_y \oplus \ell(t_x). \end{aligned}$$

As a consequence, we need to invert the matrix corresponding to the linear application mapping (t_x, t_y) to (t_u, t_v) in the expressions above. The solution is easily verified to be

$$\begin{aligned} t_x &= \ell^{-1}(t_u) \oplus t_v, \\ t_y &= t_u \oplus \ell^{-1}(t_v) \ . \end{aligned}$$

We deduce that if $u_i = x_i \oplus \ell(t_y)$ and $v_i = y_i \oplus \ell(t_x)$, then

$$\begin{aligned} x_i &= u_i \oplus \ell\left(t_u \oplus \ell^{-1}(t_v)\right) = u_i \oplus t_v \oplus \ell(t_u), \\ y_i &= v_i \oplus \ell\left(\ell^{-1}(t_u) \oplus t_v\right) = u_i \oplus \ell(t_v) \oplus t_u \ . \end{aligned}$$

□

We also remark that $\ell_w^T = \ell_w$. To see it, we simply write it as a 2×2 matrix operating on 16-bit words using $\mathcal{I}$ to denote the 16×16 identity matrix and 0 to denote the 16×16 zero matrix, and we obtain that

$$\ell = \begin{bmatrix} 0 & \mathcal{I} \\ \mathcal{I} & \mathcal{I} \end{bmatrix} \ ,$$

which is symmetric. For completeness, we provide detailed matrices of the linear Feistel functions $\mathcal{M}_2, \mathcal{M}_3, \mathcal{M}_4$ used respectively in SPARKLE256$_{n_s}$, SPARKLE384$_{n_s}$, SPARKLE512$_{n_s}$. In the following equation note that, despite the simple structure of $\mathcal{M}_{h_b}$, the matrices are rather dense.

$$\mathcal{M}_2 = \begin{bmatrix} \mathcal{I} & 0 & 0 & \mathcal{I} & 0 & 0 & 0 & \mathcal{I} \\ 0 & \mathcal{I} & \mathcal{I} & \mathcal{I} & 0 & 0 & \mathcal{I} & \mathcal{I} \\ 0 & \mathcal{I} & \mathcal{I} & 0 & 0 & \mathcal{I} & 0 & 0 \\ \mathcal{I} & \mathcal{I} & 0 & \mathcal{I} & \mathcal{I} & \mathcal{I} & 0 & 0 \\ 0 & 0 & 0 & \mathcal{I} & \mathcal{I} & 0 & 0 & \mathcal{I} \\ 0 & 0 & \mathcal{I} & \mathcal{I} & 0 & \mathcal{I} & \mathcal{I} & \mathcal{I} \\ 0 & \mathcal{I} & 0 & 0 & 0 & \mathcal{I} & \mathcal{I} & 0 \\ \mathcal{I} & \mathcal{I} & 0 & 0 & \mathcal{I} & \mathcal{I} & 0 & \mathcal{I} \end{bmatrix}, \mathcal{M}_3 = \begin{bmatrix} \mathcal{I} & 0 & 0 & \mathcal{I} & 0 & 0 & 0 & \mathcal{I} & 0 & 0 & 0 & \mathcal{I} \\ 0 & \mathcal{I} & \mathcal{I} & \mathcal{I} & 0 & 0 & \mathcal{I} & \mathcal{I} & 0 & 0 & \mathcal{I} & \mathcal{I} \\ 0 & \mathcal{I} & \mathcal{I} & 0 & 0 & \mathcal{I} & 0 & 0 & 0 & \mathcal{I} & 0 & 0 \\ \mathcal{I} & \mathcal{I} & 0 & \mathcal{I} & \mathcal{I} & \mathcal{I} & 0 & 0 & \mathcal{I} & \mathcal{I} & 0 & 0 \\ 0 & 0 & 0 & \mathcal{I} & \mathcal{I} & 0 & 0 & \mathcal{I} & 0 & 0 & 0 & \mathcal{I} \\ 0 & 0 & \mathcal{I} & \mathcal{I} & 0 & \mathcal{I} & \mathcal{I} & \mathcal{I} & 0 & 0 & \mathcal{I} & \mathcal{I} \\ 0 & \mathcal{I} & 0 & 0 & 0 & \mathcal{I} & \mathcal{I} & 0 & 0 & \mathcal{I} & 0 & 0 \\ \mathcal{I} & \mathcal{I} & 0 & 0 & \mathcal{I} & \mathcal{I} & 0 & \mathcal{I} & \mathcal{I} & 0 & 0 & 0 \\ 0 & 0 & 0 & \mathcal{I} & 0 & 0 & 0 & \mathcal{I} & \mathcal{I} & 0 & 0 & \mathcal{I} \\ 0 & 0 & \mathcal{I} & \mathcal{I} & 0 & \mathcal{I} & \mathcal{I} & \mathcal{I} & 0 & \mathcal{I} & \mathcal{I} & \mathcal{I} \\ 0 & \mathcal{I} & 0 & 0 & 0 & \mathcal{I} & 0 & 0 & 0 & \mathcal{I} & \mathcal{I} & 0 \\ \mathcal{I} & \mathcal{I} & 0 & 0 & \mathcal{I} & \mathcal{I} & 0 & 0 & \mathcal{I} & \mathcal{I} & 0 & \mathcal{I} \end{bmatrix},$$

$$\mathcal{M}_4 = \begin{bmatrix} \mathcal{I} & 0 & 0 & \mathcal{I} & 0 & 0 & 0 & \mathcal{I} & 0 & 0 & 0 & \mathcal{I} & 0 & 0 & 0 & \mathcal{I} \\ 0 & \mathcal{I} & \mathcal{I} & \mathcal{I} & 0 & 0 & \mathcal{I} & \mathcal{I} & 0 & 0 & \mathcal{I} & \mathcal{I} & 0 & 0 & \mathcal{I} & \mathcal{I} \\ 0 & \mathcal{I} & \mathcal{I} & 0 & 0 & \mathcal{I} & 0 & 0 & 0 & \mathcal{I} & 0 & 0 & 0 & \mathcal{I} & 0 & 0 \\ \mathcal{I} & \mathcal{I} & 0 & \mathcal{I} & \mathcal{I} & \mathcal{I} & 0 & 0 & \mathcal{I} & \mathcal{I} & 0 & 0 & \mathcal{I} & \mathcal{I} & 0 & 0 \\ 0 & 0 & 0 & \mathcal{I} & \mathcal{I} & 0 & 0 & \mathcal{I} & 0 & 0 & 0 & \mathcal{I} & 0 & 0 & 0 & \mathcal{I} \\ 0 & 0 & \mathcal{I} & \mathcal{I} & 0 & \mathcal{I} & \mathcal{I} & \mathcal{I} & 0 & 0 & \mathcal{I} & \mathcal{I} & 0 & 0 & \mathcal{I} & \mathcal{I} \\ 0 & \mathcal{I} & 0 & 0 & 0 & \mathcal{I} & \mathcal{I} & 0 & 0 & \mathcal{I} & 0 & 0 & 0 & \mathcal{I} & 0 & 0 \\ \mathcal{I} & \mathcal{I} & 0 & 0 & \mathcal{I} & \mathcal{I} & 0 & \mathcal{I} & \mathcal{I} & \mathcal{I} & 0 & 0 & \mathcal{I} & \mathcal{I} & 0 & 0 \\ 0 & 0 & 0 & \mathcal{I} & 0 & 0 & 0 & \mathcal{I} & \mathcal{I} & 0 & 0 & \mathcal{I} & 0 & 0 & 0 & \mathcal{I} \\ 0 & 0 & \mathcal{I} & \mathcal{I} & 0 & 0 & \mathcal{I} & \mathcal{I} & 0 & \mathcal{I} & \mathcal{I} & \mathcal{I} & 0 & 0 & \mathcal{I} & \mathcal{I} \\ 0 & \mathcal{I} & 0 & 0 & 0 & \mathcal{I} & 0 & 0 & 0 & \mathcal{I} & 0 & 0 & 0 & \mathcal{I} & 0 & 0 \\ \mathcal{I} & \mathcal{I} & 0 & 0 & \mathcal{I} & \mathcal{I} & 0 & 0 & \mathcal{I} & \mathcal{I} & 0 & 0 & \mathcal{I} & \mathcal{I} & 0 & 0 \\ 0 & 0 & 0 & \mathcal{I} & 0 & 0 & 0 & \mathcal{I} & 0 & 0 & 0 & \mathcal{I} & \mathcal{I} & 0 & 0 & \mathcal{I} \\ 0 & 0 & \mathcal{I} & \mathcal{I} & 0 & 0 & \mathcal{I} & \mathcal{I} & 0 & 0 & \mathcal{I} & \mathcal{I} & 0 & 0 & \mathcal{I} & \mathcal{I} \\ 0 & \mathcal{I} & 0 & 0 & 0 & \mathcal{I} & 0 & 0 & 0 & \mathcal{I} & 0 & 0 & 0 & \mathcal{I} & \mathcal{I} & 0 \\ \mathcal{I} & \mathcal{I} & 0 & 0 & \mathcal{I} & \mathcal{I} & 0 & 0 & \mathcal{I} & \mathcal{I} & 0 & 0 & \mathcal{I} & \mathcal{I} & 0 & \mathcal{I} \end{bmatrix}.$$

We deduce the following lemma.

Lemma 3.4.2. *The matrix representation of the function $\mathcal{M}_w$ is symmetric.*

The key properties of such linear permutations in terms of diffusion are given by the following theorem.

Theorem 3.4.1. *For all $w > 1$, $\mathcal{M}_w$ is such that:*

- *a unique active branch in the input activates all output branches,*
- *a unique active branch in the output requires that all input branches are active,*
- *if $w > 2$ and there are two active branches with indices j and k in the input, then one of the following must occur:*
  - *only the branches j and k are active in the output,*
  - *all the output branches are active except for j,*
  - *all the output branches are active except for k, or*
  - *all the output branches are active.*

Proof. We prove each point separately.

Case with 1 input. Without loss of generality, suppose that $(x_0, y_0) \neq (0, 0)$ and that $(x_i, y_i) = (0, 0)$ for all $i > 0$. Then $t_x = x_0$ and $t_y = y_0$, so that

$$u_i = x_i \oplus \ell(y_0) \quad \text{and} \quad v_i = y_i \oplus \ell(x_0) \ .$$

If $i \neq 0$, then $u_i = \ell(y_0)$ and $v_i = \ell(x_0)$. Thus, we have

$$\begin{aligned} (u_i, v_i) &= \big(\ell(y_0), \ell(x_0)\big), \text{if } i \neq 0 \ , \\ (u_0, v_0) &= \big(x_0 \oplus \ell(y_0), y_0 \oplus \ell(x_0)\big), \end{aligned}$$

so that each pair (u_i, v_i) is the output of a permutation with input (x_0, y_0) (recall that $x \mapsto x \oplus \ell(x)$ is a permutation). Since we assume $(x_0, y_0) \neq (0, 0)$, we deduce that all (u_i, v_i) are non-zero.

Case with 1 output. If w is even, the inverse of $\mathcal{M}_w$ is $\mathcal{M}_w$ itself. We can therefore reuse the same argument as above. Suppose now that w is odd. Without loss of generality, we consider that $(u_0, v_0) \neq (0,0)$ and that $(u_i, v_i) = (0,0)$ for all $i > 0$. In this case, we have $t_u = u_0$ and $t_v = v_0$, so that

$$\begin{aligned}(x_0, y_0) &= \big(u_0 \oplus v_0 \oplus \ell(u_0),\ v_0 \oplus u_0 \oplus \ell(v_0)\big),\\ (x_i, y_i) &= \big(v_0 \oplus \ell(u_0), u_0 \oplus \ell(v_0)\big), \text{if } i \neq 0\ ,\end{aligned}$$

We deduce that (x_i, y_i) is always a permutation of (u_0, v_0) for $i \neq 0$ and thus cannot be zero. It is also the case for $i = 0$. Indeed, we have $(x_0, y_0) = \big((\ell^2(u_0) \oplus v_0, u_0 \oplus \ell^2(v_0)\big)$ because $\mathrm{I} + \ell = \ell^2 = \ell^{-1}$, so that the function mapping (u_0, v_0) to (x_0, y_0) is the inverse of $(x, y) \mapsto (x \oplus \ell(y), y \oplus \ell(x))$. In particular, it is a permutation as well.

We conclude that if a unique branch is active in the output then all branches are active in the input.

Case with 2 inputs. Suppose now that $w > 2$ and that $(x_j, y_j) \neq (0,0)$, $(x_k, y_k) \neq (0,0)$ and $(x_i, y_i) = (0,0)$ otherwise. In this case, we have $t_x = x_j \oplus x_k$ and $t_y = y_j \oplus y_k$ so that

$$\begin{aligned}(u_j, v_j) &= \big(x_j \oplus \ell(y_j \oplus y_k),\ y_j \oplus \ell(x_j \oplus x_k)\big)\ ,\\ (u_k, v_k) &= \big(x_k \oplus \ell(y_j \oplus y_k),\ y_k \oplus \ell(x_j \oplus x_k)\big)\ ,\\ (u_i, v_i) &= \big(\ell(y_j \oplus y_k), \ell(x_j \oplus x_k)\big), \text{if } i \notin \{j, k\}\ .\end{aligned}$$

If $(u_i, v_i) = (0,0)$ for some $i \notin \{j,k\}$ then $(u_i, v_i) = (0,0)$ for all $i \notin \{j,k\}$ and we have both $x_j = x_k$ and $y_j = y_k$. Hence, we have $(u_j, v_j) = (x_j, y_j) \neq (0,0)$ and $(u_k, v_k) = (x_k, y_k) \neq (0,0)$, so that both branches j and k have to be active.

Finally, we suppose that $(u_i, v_i) \neq (0,0)$ for some $i \notin \{j,k\}$. In this case, we have $(u_i, v_i) \neq (0,0)$ for all $i \notin \{j,k\}$ and we cannot have both $(u_j, v_j) = (0,0)$ and $(u_k, v_k) = (0,0)$. Indeed, if it were the case then we would have

$$\begin{aligned}x_j &= \ell(y_j \oplus y_k),\ y_j = \ell(x_j \oplus x_k)\\ x_k &= \ell(y_j \oplus y_k),\ y_k = \ell(x_j \oplus x_k)\ ,\end{aligned}$$

which in turn implies $x_j = x_k$ and $y_j = y_k$, leading to $x_j = x_k = y_j = y_k = 0$ and thus to a contradiction.

□

Corollary 3.4.1. *If $w > 2$ then the differential branch number of $\mathcal{M}_w$ is 4. If $w = 2$ then the differential branch number of $\mathcal{M}_2$ is 3. As a consequence, $\mathcal{M}_2$ and $\mathcal{M}_3$ (which are used in* Sparkle256 *and* Sparkle384 *respectively) are MDS.*

Proof. If $w > 2$ then an input with one active branch will activate all $w \geq 3$ outputs. Thus, the transition weights $1 \to 1$ and $1 \to 2$ are impossible. Conversely, using that a weight 1 output pattern must yield an input with all $w \geq 3$ inputs active, we deduce that the weight transition $2 \to 1$ is impossible as well. The branching number is thus at least equal to 4. As transitions $2 \to 2$ are possible, this bound is tight.

If $w = 2$ then the transitions with weight $1 \to 1$ are impossible as one active branch is always mapped to 2 and, conversely, a unique active output branch implies 2 active ones in the input. Hence, the branching number is 3 in this case. □

3.5 On the Number of Steps

In this section, we outline the security margins depending on the number of steps of the Sparkle instances used in the AEAD and hashing schemes. Each of our schemes employs a *big* and a *slim* version of an underlying Sparkle permutation. For design simplicity, we decided to use the same number of steps for the big permutations both in the AEAD and hashing schemes, as well as the same number of steps for the slim permutations in both functionalities.

We emphasize that the security evaluation with regard to differential and linear attacks is based on the bounds obtained by the LTS and therefore the above margins are derived under the *worse-case assumption* that differential and linear trails matching our bounds exist. In other words, we did not find actual attacks on the (round-reduced) schemes that correspond to those bounds and might actually be vastly overestimating the abilities of the adversary. Especially for the case where we had to use the worse bounds corresponding to outer part to anything trails in order to be as conservative as possible (see below), we expect a much higher margin in practice.

3.5.1 In the Big Versions

We are aiming for security of SPARKLE in the sense that no distinguishers exist with both a time and data complexity lower than $2^{\frac{b}{2}}$, where b is the block size of the permutation in bits. In particular, this means that we need 6, 7, and 7 steps of SPARKLE256, SPARKLE384 and SPARKLE-512, respectively, in order to prevent differential and linear distinguishers based on the bounds of the LTS (see Tables 4.3 and 4.4). All other attacks that we evaluated covered fewer steps. For the final choice of the number of steps, we added four steps as a security margin for SPARKLE256 and SPARKLE384 (three steps for full diffusion plus one additional step), and five steps as a security margin for SPARKLE512 (since it is intended for a higher security level), thus choosing 10, 11 and 12 steps for the big instances of SPARKLE256, SPARKLE384 and SPARKLE512, respectively. The reason for adding one, resp., two additional step after the three steps for full diffusion is that we can easily afford it without impacting the actual performance of our schemes significantly, thus leading to a more conservative design. As the big versions of the permutations are used for initializing the state with a secret key in the AEAD schemes, a more conservative approach seems reasonable. It total, this gives us a security margin of 66%, 57% and 71%, respectively.

3.5.2 In the Slim Versions

The slim version of the SPARKLE permutations are designed to offer security against distinguishers in which the adversary can only control the outer part of the state when the permutation is employed in a sponge. There are different security bounds to consider depending on whether the permutation is employed in SCHWAEMM or ESCH. For the AEAD schemes SCHWAEMM, the domain separation constant that is XORed in the last block is in the inner part. Therefore, the adversary could have the ability to inject a difference in the inner part of the last padded block. In order to prevent attacks based on this possibility, we consider the bounds in Table 4.7 corresponding to the "outer part to anything" trails, i.e., where the input difference is constrained to be in the outer part only, but the output difference can be on the whole state. Note that we are using a big permutation for separating the associated data processing from the message encryption part because the adversary might be able to inject an *input* difference into the inner part of the last (padded) block through the domain separation constant. For ESCH, we consider the bounds corresponding to the outer part to outer part trails (where the rate is always $\frac{b}{2}$ because of the indirect injection). With regard to linear attacks, we use the bounds of the outer part to outer part trails given in Table 4.8 for ESCH. For SCHWAEMM, we have to use the bounds for the permutation (i.e., Table 4.4) because of the rate-whitening layer that introduces linear masks in the inner part.

Note that in ESCH, the security level to achieve with regard to differential attacks is $\frac{c}{2}+r$. The security level with regard to linear attacks in SCHWAEMM is determined by the data limit.

SPARKLE256 is only employed in our AEAD schemes. To offer a security level of 128 bit, for SPARKLE256, we need five steps to prevent differential and linear attacks. For a differential attack, 5 steps are sufficient to prevent outer part to anything differential trails with a probability under 2^{-128} (see Table 4.7). For linear attacks, this bound is obtained already after four steps assuming a data limit of 2^{64} blocks. Recall that the estimated data complexity for a linear attack is at least $1/p^2$, where p denotes the absolute correlation of the linear approximation used. Hence, we need only to ensure an upperbound of 2^{-32} on the absolute correlation of linear trails, see Table 4.4. The best distinguishers we found with regard to other attacks in this sponge settings cover fewer steps (see Section 4.3).

Sparkle384 is employed in three of our schemes. For Schwaemm256-128, we need a security level of 128 bit for the underlying permutation, restricting the user to encrypt at most 2^{64} blocks with one key. The rate of the sponge is $r = 256$ and four steps are sufficient to prevent linear and differential distinguishers according to the LTS bounds. Also, the longest of the other distinguishers we found covers no more than four steps. In Schwaemm192-192, we need a security level of 192 bit for the underlying permutation, restricting the user to encrypt at most 2^{64} blocks with one key. The rate of the sponge is $r = 192$ and also four steps are sufficient to prevent linear distinguishers according to the LTS bounds. To prevent differential distinguishers, six steps are sufficient for outer part to anything trails, and five steps for outer part to outer part trails. In Esch256, we need a security level of 256 bit for the underlying permutation with regard to differential attacks and 128 bit with regard to linear attacks (but without the restriction of processing only 2^{64} blocks of data). The rate of the sponge is $r = 192$ (because of indirect injection) and five steps are sufficient to prevent differential attacks (using the bounds for outer part to outer part trails), and four steps to prevent linear distinguishers according to the LTS bounds.

Finally, Sparkle512 is employed in two of our schemes. For Schwaemm256-256, we need a security level of 256 bit for the underlying permutation, restricting the user to encrypt at most 2^{128} blocks with one key. The rate of the sponge is $r = 256$ and seven steps are sufficient to prevent differential distinguishers according to the outer part to anything bounds. Note that five steps are sufficient if the output difference is in the outer part only. With regard to linear attacks, five steps are sufficient. Also, the longest of the other distinguishers we found covers no more than four steps. In Esch384, we need a security level of 320 bit for the underlying permutation with regard to differential attacks and 192 bit with regard to linear attacks (but without the restriction of processing only 2^{128} blocks of data). The rate of the sponge is $r = 256$ (because of indirect injection) and we need six steps to prevent differential distinguishers and five to prevent linear distinguishers, respectively, according to the LTS bounds.

3.5.3 On the Differential and Linear Bounds

Our arguments rely on the bounds on the differential probability and the absolute linear correlation obtained by applying a long-trail argument using the properties of our ARX-box Alzette. These are conservative bounds: while our algorithms show that there *cannot exist* any trail with higher a probability/correlation, it may very well be that the bounds they find are not tight. In other words, while we cannot overestimate the security our permutations provide against single trail differential and linear attacks, we may actually *underestimate* it.

In fact, we think it is an interesting open problem to try and tighten our bounds as it could only increase the trust in our algorithms. This tightening could happen at two levels: first, we could try and obtain tighter bounds for the differential and linear properties of Alzette alone and, second, we could look for actual trails for the Sparkle permutations. Indeed, our bounds for Sparkle assume that there exists a trail where all transitions have optimal probability that fits in every truncated trail. Again, this is a conservative estimate. We may be *overestimating* the power the actual trails give to the adversary.

In both the differential and the linear case, we made some experiments at the ARX-box level to try and estimate if there was any significant clustering of trails that might lead the differential probability (respectively absolute correlation) to be significantly higher than the differential trail probability (resp. trail absolute correlation). These are described in Section 3.3.10.1 and 3.3.10.2 respectively. In the differential case, this effect is minimal. In the linear case, it is small but observable. However, in double iterations of Alzette, it is not sufficient that the input and output patterns are known, we also need to constrain the values in the middle (i.e. in between the two Alzette instances). As a consequence, we use the linear trail correlation bound and not a bound that would take the double ARX-box level clustering into account.

4 Security Analysis

4.1 Security Claims

Our proposed algorithms are secure to the best of our knowledge. We have done our best not to introduce any flaw in their design. In particular, we did not purposefully put any backdoor or other security flaw in our algorithms.

4.1.1 For Esch

We claim that Esch256 and Esch384 offer a security level of $\frac{c}{2}$ bits, where c is both the capacity and digest size, with regard to collision resistance, preimage resistance and second preimage resistance. Our claim covers the security against length-extension attacks. We impose the data limit of $2^{\frac{c}{2}}$ processed blocks (as collisions are likely to occur for more data). In other words, a cryptanalytic result that qualifies as an attack violating the above security claim should have a time complexity of at most $2^{\frac{c}{2}}$ executions of the underlying permutation or its inverse.

For the XOFs, the security level is $\min\{\frac{c}{2}, \frac{t}{2}\}$ bits for collision resistance and $\min\{\frac{c}{2}, t\}$ bits for (second) preimage resistance. The maximal allowed output length t is the same as the data limit.

4.1.2 For Schwaemm

The Beetle mode of operation offers a security level (in bits) of $\min(r, (r+c)/2, c - \log_2(r))$ both for confidentiality (under adaptive chosen-plaintext attacks) and integrity (under adaptive forgery attempts), where r denotes the rate, and c denotes the capacity. Note that we claim security in the *nonce-respecting setting*, i.e., in which the adversary cannot encrypt data using the same nonce twice.

Following the security bound of the Beetle mode and the choice of parameter in our AEAD schemes, we claim a security level of 120 bits for Schwaemm256-128, where the adversary is allowed to process at most 2^{68} bytes of data (in total) under a single key. In other words, a cryptanalytic result that qualifies as an attack violating this security claim has a time complexity of at most 2^{120} executions of the underlying permutation or its inverse and requires at most 2^{68} blocks of data.

Analogously, we claim a security level of 184 bits for Schwaemm192-192, where the adversary is allowed to process at most 2^{68} byte of data under a single key. For Schwaemm128-128 we claim a security level of 120 bits,[1] where the adversary is allowed to process at most 2^{68} byte of data under a single key. Finally, for Schwaemm256-256 we claim a security level of 248 bits, where the adversary is allowed to process at most 2^{133} byte of data under a single key.[2]

Nonce Misuse Setting. The above security claims are void in cases where nonces are resused. As noted in [VV17], authenticated ciphers based on duplexed sponge constructions are vulnerable to CPA decryption attacks and semi-universal forgery attacks in the nonce-misuse setting. For instance, an encryption of $M = 0^r$ under (K, N, A) leaks the outer part of the internal state for processing the first message block as the ciphertext. This information can be used to decrypt another ciphertext obtained under the same tuple (K, N, A).

However, because we employ a strong permutation in the initialization and a permutation in the absorption that is strong when the adversary can only control the outer part, we believe that a full *state recovery attack* is hard to mount in practice even in the nonce-misuse setting. We therefore expect some reasonable security against key recovery attacks even under reuse of nonces. Moreover, we expect the security with regard to privacy and integrity to still hold under misuse

[1] While Beetle allows us to claim 121 bits, we claim only 120 in order to have a security level consistent with Schwaemm256-128.

[2] We first set the data limit as 2^{64} blocks for all instances except for Schwaemm256-256 but then decided to replace 2^{69} bytes by 2^{68} to have more consistency between the algorithms.

of nonces if it is *guaranteed* that the associated data is *unique* for each encryption. In that way, the associated data itself would serve as a nonce before invoking the encryption process of the secret messages. We emphasize that this statement about nonce misuse robustness is not part of our formal security claim and therefore, we *strongly* recommend to only use the algorithm under unique nonces.

Known-key Attacks. In the secret-key setting, Beetle guarantees security level close to the capacity size. However, in the known-key setting, the security drops to half of the capacity size. Indeed, a classical meet-in-the-middle attack in the sponge becomes possible. It allows an attacker to find collisions and preimages (in the case where the tag is squeezed in one step) with birthday complexity. We stress that such attacks are possible in all sponge-based modes.

We are not aware of usage scenarios of authenticated encryption which require known-key security. Therefore, we do not claim any known-key security of the SCHWAEMM family.

4.1.3 For Sparkle

For SPARKLE256_{10}, we claim that there are no distinguishers with both a time and data complexity lower than 2^{128}. For SPARKLE384_{11}, we claim that there are no distinguishers with both a time and data complexity lower than 2^{192}. For SPARKLE512_{12}, we claim that there are no distinguishers with both a time and data complexity lower than 2^{256}.

A Remark on the Slim Versions. The slim version of the SPARKLE permutations are designed to offer security against distinguishers in which the adversary can only control part of the state, in particular the part corresponding to the rate r when the permutation is employed in a sponge. We emphasize that those slim versions should *not* be used on their own or in other constructions that are not the sponge-based ones presented in this paper unless a proper security analysis is done.

4.1.4 Targets for Cryptanalysis

We encourage cryptanalysts to study variants of ESCH and SCHWAEMM with fewer steps or a decreased capacity. Particularly, for hashing, we define the targets (s,b)-ESCH256 and (s,b)-ESCH384, which instantiate a version of ESCH256, resp., ESCH384 using s steps in the slim and b steps in the big permutation.

Similarly, for authenticated encryption, we define the members (s,b)-SCHWAEMMr-c, which instantiates a version of SCHWAEMMr-c using s steps in the slim and b steps in the big permutation. Note that those additional members are not part of our submission, but just define possible targets for cryptanalysis.

We also encourage the study of variants of our ARX-box Alzette with different round constants.

4.2 Attacks and Tests Against the Permutation

We evaluated the security of the SPARKLE permutation family against several attack vectors, listed in Table 4.1, with regard to our security claim stated above. Note that in the attacks considered here the adversary can control the whole permutation state, not only the part corresponding to the rate when the permutation is used in a sponge.

Besides the attacks listed in Table 4.1, we conducted several statistical tests on the SPARKLE permutations. In particular, we tested for diffusion properties (Section 4.2.1), the distribution of low-weight monomials in the algebraic normal form (Section 4.2.8.1) and the resistance against slide attacks (Section 4.2.9).

4.2.1 Diffusion Test (SAC)

The *strict avalanche criterion (SAC)* was introduced in [WT86] as a measure for the diffusion in cryptographic functions. The SAC states that, for each input bit i and output bit j, whenever the i-th input bit is flipped, the j-th output bit should flip with probability $\frac{1}{2}$. As proposed in [WT86], we took several random samples and computed the $n \times n$ *dependence matrix* $\mathbb{D}$, where n denotes

Table 4.1: This table lists upper bounds on the number of steps for which we found an attack, or for which the differential and linear bounds are too low to guarentee security, breaking a security level of $\frac{b}{2}$ bits, where b denotes the block size, with regard to several attack vectors. The numbers for differential and linear attacks correspond to the bounds given in Table 4.3 and Table 4.4.

Attack	**Ref.**	SPARKLE256	SPARKLE384	SPARKLE512
Differential cryptanalysis	[BS91]	4	5	6
Linear cryptanalysis	[Mat94]	5	6	6
Boomerang attacks	[Wag99]	3	4	5
Truncated differentials	[Knu95]	2	2	3
Yoyo games	[BBD+99]	4	4	4
Impossible differentials	[Knu98, BBS99]	4	4	4
Zero-correlation	[BR14]	4	4	4
Integral and Division property	[DKR97, KW02, Tod15]	4	4	4
	# steps slim	7	7	8
	# steps big	10	11	12

the block size of the cryptographic permutation in bits. Let S denote the set of n-bit random samples and F the permutation to test. Then, the entries of the dependence matrix are defined as $\mathbb{D}_{i,j} = |\{x \in S \mid F(x)_j \neq F(x \oplus 2^i)_j\}|$. For a secure cryptographic permutation, we expect that each $\mathbb{D}_{i,j}$ is a binomial distributed random variable following $B(|S|, \frac{1}{2})$.

We evaluated the dependence matrix of SPARKLE for all block sizes n and for several numbers of steps n_s. Results for 10^7 random samples are given in Table 4.2. Figure 4.1 depicts the distribution of the occurrences of certain values in $\mathbb{D}_{i,j}$ for SPARKLE384$_3$ and 10^6 random samples, compared to what one would expect from a binomial distribution. Note that, for all SPARKLE variants, full diffusion is reached after three steps. Our results indicate that three steps are also enough to fulfill the SAC.

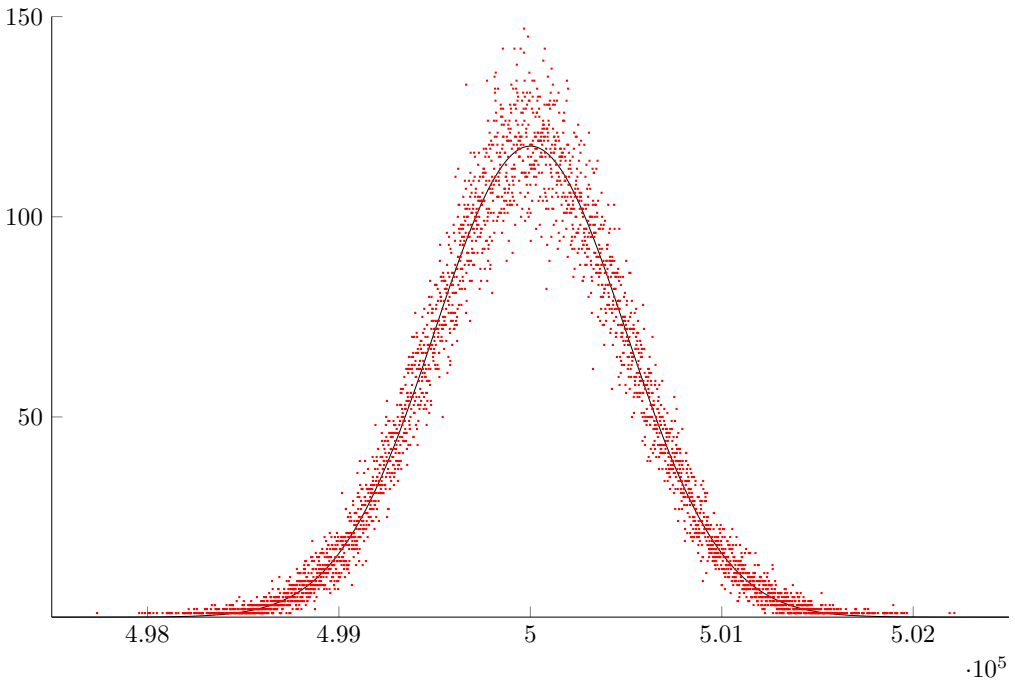


Figure 4.1: Number of times (y axis) for which a specific entry (x axis) occurs in the dependency matrix after three steps of SPARKLE384 for 10^6 samples (red) and the theoretical number of occurrences corresponding to a $B(10^6, \frac{1}{2})$ distribution (black).

Table 4.2: Properties of the dependency matrix $\mathbb{D}$ for Sparkle with block size n and number of steps n_s.

(n, n_s)	$\min_{i,j}\{\mathbb{D}_{i,j}\}$	$\max_{i,j}\{\mathbb{D}_{i,j}\}$	$\mu = \frac{1}{n^2}\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\mathbb{D}_{i,j}$	$\frac{1}{n^2}\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}(\mathbb{D}_{i,j}-\mu)^2$
$(256, 3)$	4,993,280	5,007,481	4,999,991	2,520,767
$(256, 4)$	4,992,587	5,006,115	5,000,000	2,483,369
$(256, 5)$	4,993,249	5,007,111	4,999,994	2,509,529
$(256, 6)$	4,993,593	5,007,685	4,999,997	2,499,786
$(256, 7)$	4,992,626	5,006,872	4,999,999	2,485,584
$(384, 3)$	4,993,170	5,007,839	5,000,000	2,496,373
$(384, 4)$	4,993,344	5,007,802	5,000,003	2,510,058
$(384, 5)$	4,992,917	5,007,030	4,999,995	2,504,939
$(384, 6)$	4,993,238	5,006,775	5,000,003	2,506,934
$(384, 7)$	4,992,030	5,006,687	4,999,999	2,514,274
$(512, 3)$	4,993,031	5,006,999	4,999,998	2,504,774
$(512, 4)$	4,992,929	5,007,068	4,999,999	2,501,413
$(512, 5)$	4,992,762	5,007,492	4,999,999	2,497,702
$(512, 6)$	4,992,860	5,007,736	5,000,000	2,506,286
$(512, 7)$	4,992,525	5,006,852	5,000,003	2,503,587
$(512, 8)$	4,992,790	5,007,753	5,000,000	2,492,782

4.2.2 Differential Attacks

4.2.2.1 Bounding the MEDCP

In order to bound the MEDCP for the whole permutation, we use a long trail argument. First, we enumerate all the truncated trails defined at the branch level (i.e., a branch is active or inactive) that are compatible with the linear layer. For each such truncated trail, we partition it into long trails and then deduce a bound on the probability of all the trails that fit into this truncated using the probability that were established for Alzette (see Section 3.3.3).

In practice, a truncated trail is a sequence of binary vectors d_i of length n_b where, in our case, $n_b \in \{4, 6, 8\}$. Furthermore, the structure of the linear layer significantly constrains the value of d_{i+1} knowing d_i. Indeed, half of the bits have to be identical (corresponding to those that do not receive a XOR), and the output of the Feistel function itself is constrained by Theorem 3.4.1. As a consequence, we can implement the exploration of all truncated trails as a tree search with the knowledge that each new step will multiply the number of branches at most by $2^{n_b/2} \in \{8, 16, 32\}$.

We can simplify the search further using tricks borrowed from Matsui's algorithm 1. We can for example fix a threshold for the probability of the differential trail we are interested in, thus allowing us to cut branches as soon as the probability of the trails they contain is no longer bounded by the given threshold. If this threshold is higher than the actual bound, the search will return a correct result and it will do so faster.

We have implemented the long trail argument in this way to bound the differential probability in a permutation with the structure used in the Sparkle family. The bounds for the permutations we have obtained using the bounds for our 4-round Alzette instances are given in Table 4.3.

Table 4.3: The quantity $-\log_2(p)$ where p is the differential bound for several steps of Sparkle for different block sizes. For 1 and 2 steps, we always have that $-\log_2(p)$ is equal to 6 and 32 respectively.

$n \setminus$ steps	3	4	5	6	7	8	9	10	11	12	13
256	64	88	140	168	192	216	≥ 256	≥ 256	≥ 256	≥ 256	≥ 256
384	70	100	178	200	230	260	326	356	≥ 384	≥ 384	≥ 384
512	76	112	210	232	268	276	295	424	433	496	≥ 512

4.2.2.2 Truncated Differential Trails

We performed an exhaustive search of all truncated trails on the branch level, i.e., each branch can be either active or inactive. We define the *weight* of a truncated differential trail as $\frac{-\log_2(p)}{64}$, where p denotes its probability. The truncated differential is said to be *effective* if its weight is strictly smaller than the number of inactive words in the output. Our approach consists of two stages. The first stage is to generate the matrix of probabilities of all truncated transitions through the linear layer. We propose a new generic and precise method for this step. The second stage is a simple iterative search, where for each round and for each truncated pattern at this round we keep the best truncated trail leading tho this pattern.

Generating the Truncated Trail Matrix of a Linear Layer. Recently, a MILP approach for searching for truncated differential trails was proposed in [MA20]. The method uses the matrix of probabilities of truncated transitions through a linear layer, called the branching property table. However, the precise method to compute this table was not described there.

We describe a generic method to generate the matrix of probabilities of truncated transitions from the binary matrix of the analyzed linear layer.

Let $L : (\mathbb{F}_2^m)^t \to (\mathbb{F}_2^m)^t$ be a linear bijective mapping. An *exact truncated transition* over L is a pair of vectors from $\{0, *\}^t$. A *loose truncated transition* over L is a pair of vectors from $\{0, \diamond\}^t$. A truncated transition α, β over L is denoted $\alpha \xrightarrow{L} \beta$.

For $\gamma_i \in \{0, \diamond, *\}$ let $p(\gamma_i)$ be defined as

$$p(\gamma_i) = \begin{cases} \{0\}, & \text{if } \gamma_i = 0, \\ \mathbb{F}_2^m, & \text{if } \gamma_i = \diamond, \\ \mathbb{F}_2^m \setminus \{0\}, & \text{if } \gamma_i = *, \end{cases}.$$

For a vector $\gamma \in \{0, \diamond, *\}^t$ let $p(\gamma)$ denote the set $p(\gamma_0) \times \ldots \times p(\gamma_{t-1})$.

The *cardinality* of a truncated transition $\alpha \xrightarrow{L} \beta$ is defined as

$$|\alpha \xrightarrow{L} \beta| = |\{x \in p(\alpha) \mid L(x) \in p(\beta)\}| = |L(p(\alpha)) \cap p(\beta)|\,.$$

The *probability* of a truncated transition $\alpha \xrightarrow{L} \beta$ is defined as

$$\Pr[\alpha \xrightarrow{L} \beta] = \Pr_{x \in p(\alpha)}[L(x) \in p(\beta)] = \frac{|\alpha \xrightarrow{L} \beta|}{p(\alpha)}.$$

Since p is easy to compute, we focus on computing the cardinality of an exact truncated transition.

The first step is to compute the cardinalities of all possible *loose* truncated transitions over L. Let $\alpha \xrightarrow{L} \beta$ be a loose transition over L. Observe that $p(a), p(b)$ are linear subspaces of $(\mathbb{F}_2^m)^t$. The cardinality $|\alpha \xrightarrow{L} \beta| = L(p(\alpha)) \cap p(\beta)$ can be computed using basic linear algebra as follows. First, we choose a basis of $p(\alpha)$ and map it through L to obtain a basis of $L(p(\alpha))$, which we denote B. A vector b belongs to $p(\beta)$ if and only if $b_i = 0$ when $\beta_i = 0$. Let $\pi_\beta(b)$ be the part of the vector b consisting of all elements b_i for which $\beta_i = 0$. We compute $r = \mathrm{Rank}(\pi_\beta(B))$ and conclude that precisely $2^{\dim p(\alpha) - r}$ elements of $L(p(\alpha))$ belong to $p(\beta)$, i.e. $|\alpha \xrightarrow{L} \beta| = 2^{\dim p(\alpha) - r}$.

The second step is to compute the probabilities of all *exact* truncated transitions over L. Indeed, a loose truncated truncated trail can be seen as a union of precise truncated trails. For example,

$$(\diamond, 0) \xrightarrow{L} (0, \diamond)$$

is equivalent to a union of disjoint transitions

$$\{(0,0) \xrightarrow{L} (0,0), \quad (0,0) \xrightarrow{L} (0,*), \quad (*,0) \xrightarrow{L} (0,0), \quad (*,0) \xrightarrow{L} (0,*)\},$$

i.e. the cardinalities are summed.

Formally, for any $\gamma, \delta \in \{0, \diamond, *\}^t$ let $\gamma \preceq \delta$ if and only if $\delta_i \preceq \gamma_i$ for all $0 \le i < t$, where $\delta_i \preceq \gamma_i$ if and only if $\delta_i = 0$ or $\gamma_i \neq 0$.

Let $\alpha_* \xrightarrow{L} \beta_*$ be an exact truncated transition and let $\alpha_\diamond \xrightarrow{L} \beta_\diamond$ be the same transition, with $*$ replaced by $\diamond$, i.e. the loose variant of $\alpha_* \xrightarrow{L} \beta_*$. Clearly, the loose transition $\alpha_\diamond \xrightarrow{L} \beta_\diamond$ can be partitioned into disjoint exact transitions. Let

$$S = \left\{ \alpha' \xrightarrow{L} \beta' \mid \alpha' \in \{0,*\}^t, \alpha' \preceq \alpha_*, \beta' \in \{0,*\}^t, \beta' \preceq \beta_* \right\}.$$

Then

$$|\alpha_\diamond \xrightarrow{L} \beta_\diamond| = \sum_{\alpha' \xrightarrow{L} \beta' \in S} |\alpha' \xrightarrow{L} \beta'|.$$

It follows that

$$|\alpha_* \xrightarrow{L} \beta_*| = |\alpha_\diamond \xrightarrow{L} \beta_\diamond| - \sum_{\alpha' \xrightarrow{L} \beta' \in S, (\alpha',\beta') \neq (\alpha_*,\beta_*)} |\alpha' \xrightarrow{L} \beta'|. \tag{4.1}$$

Note that $|\alpha_* \xrightarrow{L} \beta_*|$ can be also expressed in terms of cardinalities of loose "subtransitions" of $\alpha_\diamond \xrightarrow{L} \beta_\diamond$, which were computed in the first step, by using the inclusion-exclusion principle. However, since we need to compute all cardinalities of all exact transitions, we can simply use the *partition* into exact "subtransitions" given by Equation 4.1. Indeed, by computing the cardinalities in the lexicographic order of transitions, we can ensure that all sub-transitions are processed before processing the current transition.

Given the cardinalities of exact transitions, it is easy to compute the probabilities of exact transitions and compute the truncated trail matrix.

The time complexity of the naive implementation is $\mathrm{O}(4^t \cdot ((tm)^3 + 4^t))$. The sum over "subtransitions" can be done in one extra pass in time $\mathrm{O}(t \cdot 4^t)$ using dynamic programming. Then the complexity becomes fully dominated by the linear algebra: $\mathrm{O}(4^t \cdot (tm)^3)$.

Truncated Trails in Sparkle. We consider the linear layer of Sparkle as a mapping of $(\mathbb{F}_2^{64})^{n_b}$ to itself, and search for all truncated differentials on the high level. The truncated differential is said to be *effective* if its weight is strictly smaller than the number of inactive words in the output. For Sparkle256, the longest effective truncated differential trail covers two steps and has weight 0. It can be described as follows, where $*$ indicates an active branch and 0 indicates an inactive branch:

$$\begin{array}{lcccc} \text{input}: & 0 & 0 & 0 & * \\ \text{step 1}: & * & 0 & 0 & 0 \\ \text{step 2}: & * & * & * & 0 \end{array}.$$

Another similar one can be obtained using the input $00*0$. When restricting the input difference to be only in the left branches (i.e., for the setting in Schwaemm128-128), the longest effective truncated differential trail covers only one step and has weight 0:

$$\begin{array}{lcccc} \text{input}: & * & 0 & 0 & 0 \\ \text{step 1}: & * & * & * & 0 \end{array}.$$

For Sparkle384, the longest effective truncated differential trail also covers two steps and has weight 0. It is also valid for the setting in Schwaemm256-128:

$$\begin{array}{lcccccc} \text{input}: & 0 & 0 & 0 & * & 0 & 0 \\ \text{step 1}: & 0 & 0 & * & 0 & 0 & 0 \\ \text{step 2}: & * & * & * & 0 & 0 & * \end{array}.$$

Two similar ones can be obtained using inputs $0000*0$ and $00000*$.

For Sparkle512, the longest effective truncated differential trail covers three steps and has weight 1. It cannot be used in the setting of Schwaemm256-256:

$$\begin{array}{lcccccccc} \text{input}: & 0 & 0 & 0 & 0 & 0 & * & 0 & * \\ \text{step 1}: & * & 0 & * & 0 & 0 & 0 & 0 & 0 \\ \text{step 2}: & 0 & * & 0 & * & * & 0 & * & 0 \\ \text{step 3}: & * & * & * & * & 0 & * & 0 & * \end{array},$$

where we associate a probability of 2^{-64} for the transition between step 1 and step 2.

Such truncated trails where two branches are active and activate only two outputs of the Feistel functions exist for all number of branches strictly greater than 4 and, in particular, for both SPARKLE384 and SPARKLE512.The purpose of the rotation of the branches after the addition of the Feistel function in the linear layer is to prevent the iteration of such truncated trails. While they would pose no threat as such, they could serve as the template for high probability "not-truncated" differential trails.

Overall, truncated trails do not threaten the security of any version of SPARKLE.

4.2.3 Linear Attacks

For linear attacks, we can use essentially the same analysis as for differential attacks. The only difference is that we need to replace the linear layer of SPARKLE by the transpose of its inverse. The linear layer can be written as $\mathcal{L}_{n_b} = \mathcal{F} \times \mathcal{R} \times \mathcal{S}$ where $\mathcal{F}$ corresponds to the Feistel function, $\mathcal{R}$ to the rotation applied to the branches on the right side and $\mathcal{S}$ to the swap of the branches on the left and right sides. For example, for SPARKLE384, they correspond to the following block matrices:

$$\mathcal{S} = \begin{bmatrix} & & & \mathcal{I} & & \\ & & & & \mathcal{I} & \\ & & & & & \mathcal{I} \\ \mathcal{I} & & & & & \\ & \mathcal{I} & & & & \\ & & \mathcal{I} & & & \end{bmatrix}, \mathcal{R} = \begin{bmatrix} \mathcal{I} & & & & & \\ & \mathcal{I} & & & & \\ & & \mathcal{I} & & & \\ & & & \mathcal{I} & & \\ & & & & \mathcal{I} & \\ & & & & & \mathcal{I} \end{bmatrix}$$

and

$$\mathcal{F} = \begin{bmatrix} \mathcal{I} & & & & & \\ & \mathcal{I} & & & & \\ & & \mathcal{I} & & & \\ \mathcal{I}+\ell' & \ell' & \ell' & \mathcal{I} & & \\ \ell' & \mathcal{I}+\ell' & \ell' & & \mathcal{I} & \\ \ell' & \ell' & \mathcal{I}+\ell' & & & \mathcal{I} \end{bmatrix},$$

where $\ell'(x, y) = \ell(y), \ell(x)$. We thus have that $(\mathcal{L}_{n_b}^T)^{-1} = (\mathcal{F}^T)^{-1} \times (\mathcal{R}^T)^{-1} \times (\mathcal{S}^T)^{-1}$. We can simplify this expression using that:

- $\mathcal{F}$ is an involution, so that $(\mathcal{F}^T)^{-1} = \mathcal{F}^T$,
- $\mathcal{R}$ is orthogonal, so that $(\mathcal{R}^T)^{-1} = \mathcal{R}$, and
- $\mathcal{S} = \mathcal{S}^{-1} = \mathcal{S}^T$, so that $(\mathcal{S}^T)^{-1} = \mathcal{S}$.

We deduce that $(\mathcal{L}_{n_b}^T)^{-1} = (\mathcal{F}^T) \times \mathcal{R} \times \mathcal{S}$. The permutation $\mathcal{F}^T$ has the following representation

$$\mathcal{F}^T = \begin{bmatrix} \mathcal{I} & & \mathcal{I}+\ell'^T & \ell'^T & \ell'^T \\ & \mathcal{I} & \ell'^T & \mathcal{I}+\ell'^T & \ell'^T \\ & & \mathcal{I} & \ell'^T & \mathcal{I}+\ell'^T \\ & & & \mathcal{I} & \\ & & & & \mathcal{I} \end{bmatrix},$$

i.e. it is a Feistel function going from the right to the left and where $\mathcal{M}_w$ is replaced with its transpose. As we established in Lemma 3.4.2, this transpose is in fact $\mathcal{M}_w$ itself.

As a consequence, we can reuse the long trail argument that we introduced for the differential case (see Section 4.2.2.1) and we can further reuse the corresponding algorithm. However, we need to modify it so that the Feistel function goes in the other direction. The bound on the maximum expected absolute linear trail correlation we obtained with this modified program is given in Table 4.4.

As argued in Section 3.5.3, the slight clustering observed in the double iteration of Alzette does not mean that the absolute correlation of trails is a bad approximation of the absolute correlations

in the permutation. Indeed, in each double iteration of Alzette, either the mask in the middle enters the linear Feistel function or it is XORed with another mask. Hence, the clustering at the double ARX-box level does not really matter (especially as it is low in our case), it is the one at the single ARX-box level (which is negligible here).

Table 4.4: The quantity $-\log_2(p)$ where p is the linear bound for several steps of Sparkle for different block sizes. For 1 and 2 steps, we always have that $-\log_2(p)$ is equal to 2 and 17 respectively.

$n \setminus$ steps	3	4	5	6	7	8	9	10	11	12	13
256	23	42	57	72	91	106	125	≥ 128	≥ 128	≥ 128	≥ 128
384	25	46	76	89	110	131	161	174	≥ 192	≥ 192	≥ 192
512	27	50	93	106	129	152	195	208	231	254	≥ 256

4.2.4 Boomerang Attacks

In Boomerang distinguishers [Wag99], the permutation P under attack is split into two parts, i.e., $P = P_1 \circ P_0$ and differentials $\alpha \xrightarrow{P_0} \beta$ for and $\gamma \xrightarrow{P_1} \delta$ with high probability are exploited. In particular, if $\alpha \xrightarrow{P_0} \beta$ holds with probability p and $\gamma \xrightarrow{P_1} \delta$ holds with probability q, a Boomerang differential with probability p^2q^2 could be constructed. On ARX-box level, Boomerang attacks look indeed dangerous. Suppose we have two differentials for one Alzette instance with probability 2^{-6}, one can construct a Boomerang differential over *two* iterations of Alzette with probability $2^{-4\cdot 6} = 2^{-24}$, which is larger than the upper bound on the maximum expected differential trail probability of 2^{-32}.

Fortunately, for the whole permutation, classical Boomerang attacks seem not to be a threat (using the bounds given in Table 4.3). Furthermore and more importantly, it is not clear how the existence of a boomerang distinguisher could be a problem in our sponge-based setting.

4.2.5 Yoyo Games

Yoyo distinguishers were first introduced in [BBD+99] where they were used to highlight the poor diffusion in the block cipher Skipjack. We have also looked experimentally at yoyos. We found several probability 1 truncated yoyos covering 4 steps of all Sparkle instances. However, none of them covers more than 4 steps. Those that do all have the same structure, which we describe using "plaintext" and "ciphertext" to denote the input and output of the permutation (even though there is no key in the permutation):

1. a difference over one of the right branches is introduced in the "plaintext" to create a second one,
2. the permutation is called on each to obtain two "ciphertexts",
3. two new "ciphertexts" are created by swapping one of the left branch between the two previously obtained "ciphertexts". Then, finally,
4. we call the inverse permutation on these new "ciphertexts".

The two new "plaintexts" have a difference equal to 0 on all but one branch, a property which should not hold for a random permutation. Besides, as for boomerangs, it is not clear that yoyos could be leveraged in our sponge-based setting if they were possible.

4.2.6 Impossible Differentials

Although the step structure is very close to a Feistel structure, the well known impossible differential for 3-round and 5-round Feistel networks does not hold for 3 steps, resp., 5 steps of Sparkle.

However, another family of impossible truncated differential exists that covers 3 steps and which leverages the properties of $\mathcal{M}_w$ described in Theorem 3.4.1.

The longest truncated impossible differential we found using MILP covers four steps. We used the approach described in [CCF+21, ST17] on the truncated level by considering Alzette as a black box. For Sparkle256, one of the longest so-obtained impossible differential is as follows, where $\alpha \neq 0, \beta \neq 0$ denote arbitrary 64-bit differences:

$$\begin{array}{lccccc} \text{input}: & 0 & 0 & 0 & \alpha & \downarrow \\ \text{step 1}: & \gamma & 0 & 0 & 0 & \downarrow \\ \text{step 2}: & & & \lightning & & \\ \text{step 3}: & 0 & 0 & \delta & 0 & \uparrow \\ \text{step 4}: & 0 & \beta & 0 & 0 & \uparrow \end{array}.$$

Similarly, for Sparkle384, a four-step impossible differential is given by

$$\begin{array}{lccccccc} \text{input}: & 0 & 0 & 0 & \alpha & 0 & 0 & \downarrow \\ \text{step 1}: & 0 & 0 & \gamma & 0 & 0 & 0 & \downarrow \\ \text{step 2}: & & & & \lightning & & & \\ \text{step 3}: & 0 & 0 & 0 & \delta & 0 & 0 & \uparrow \\ \text{step 4}: & 0 & 0 & \beta & 0 & 0 & 0 & \uparrow \end{array}.$$

Also, for Sparkle512, a four-step impossible differential is given by

$$\begin{array}{lcccccccccc} \text{input}: & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \alpha & \downarrow \\ \text{step 1}: & 0 & 0 & \gamma & 0 & 0 & 0 & 0 & 0 & \downarrow \\ \text{step 2}: & & & & & & \lightning & & & \\ \text{step 3}: & 0 & 0 & 0 & 0 & \delta & 0 & 0 & 0 & \uparrow \\ \text{step 4}: & 0 & 0 & 0 & \beta & 0 & 0 & 0 & 0 & \uparrow \end{array}.$$

Since the diffusion in Alzette is very good, we do not expect that taking its specifics into account will yield significantly better results.[3]

4.2.7 Zero-Correlation Attacks

With the same approach as for impossible differentials, we evaluated the security against truncated zero-correlation distinguishers by considering the transpose of the inverse of the linear layer. For all Sparkle versions, the longest distinguisher we found in that way also covers four steps. One of them can be given as follows, where $\alpha \neq 0$, $\beta \neq 0$ denote arbitrary masks on the 64-bit word.

For Sparkle256:

$$\begin{array}{lccccc} \text{input}: & 0 & \alpha & 0 & 0 & \downarrow \\ \text{step 1}: & 0 & 0 & 0 & \gamma & \downarrow \\ \text{step 2}: & & & \lightning & & \\ \text{step 3}: & 0 & \delta & 0 & 0 & \uparrow \\ \text{step 4}: & 0 & 0 & 0 & \beta & \uparrow \end{array}.$$

For Sparkle384:

$$\begin{array}{lccccccc} \text{input}: & 0 & 0 & \alpha & 0 & 0 & 0 & \downarrow \\ \text{step 1}: & 0 & 0 & 0 & 0 & 0 & \gamma & \downarrow \\ \text{step 2}: & & & & \lightning & & & \\ \text{step 3}: & \delta & 0 & 0 & 0 & 0 & 0 & \uparrow \\ \text{step 4}: & 0 & 0 & 0 & \beta & 0 & 0 & \uparrow \end{array}.$$

For Sparkle512:

$$\begin{array}{lcccccccccc} \text{input}: & 0 & 0 & 0 & \alpha & 0 & 0 & 0 & 0 & \downarrow \\ \text{step 1}: & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \gamma & \downarrow \\ \text{step 2}: & & & & & & \lightning & & & \\ \text{step 3}: & 0 & 0 & 0 & \delta & 0 & 0 & 0 & 0 & \uparrow \\ \text{step 4}: & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \beta & \uparrow \end{array}.$$

[3] Note that those impossible differentials can be trivially extended for one more Alzette layer, but not for a whole step.

4.2.8 Integral and Division Property (or Algebraic Degree)

4.2.8.1 Distribution of Low-Weight Monomials

As was done for KECCAK, we conducted experiments on the distribution of low-degree monomials for (round-reduced versions of) the SPARKLE permutations. For a random permutation on n bit, we would expect that, for each output bit, the number of monomials of fixed degree $d < n$ in its ANF can be approximated by a Binomial distribution $B(\binom{n}{d}, \frac{1}{2})$. In Table 4.5, for each SPARKLE variant up to three steps, we denote the maximum $d \in \{0, \ldots, 16\}$ such that the following experiment passes for all $d' \leq d$:

We choose N monomials of degree d uniformly at random and compute for each output bit y_i how many of those monomials occur in its ANF. Let us denote this number by N_{y_i}. The experiment passes if, for all y_i, $N_{y_i} \in [\frac{N}{2} - a, \frac{N}{2} + a]$, where a is the minimum number such that the probability of N_{y_i} being in $[\frac{N}{2} - a, \frac{N}{2} + a]$ is $\geq 1 - 0.00001$ assuming the Binomial distribution. For example, when choosing $N = 10000$ samples, the test passes if and only if, for all y_i, we observe $4779 \leq N_{y_i} \leq 5221$.[4]

For all SPARKLE variants, the monomial test passes for all $d \leq 16$ already after three steps.

Table 4.5: For each of the SPARKLE versions, the table contains the maximum degree d such that the monomial test passes for all $d' \leq d$. We tested up to $d = 16$.

steps	SPARKLE256	SPARKLE384	SPARKLE512
1	0	0	0
2	0	0	0
3	≥ 16	≥ 16	≥ 16

4.2.8.2 Division Property of the Sparkle Permutations

We performed MILP-aided bit-based division property analysis [Tod15, SWW20] on the SPARKLE permutation family.

For the MILP encoding of the linear layer we used the original method from [SWW20]. Note that in [ZR19] it was shown that this method is imprecise and may result in extra trails and weaker distinguisher. The linear layer of SPARKLE can be viewed as 16 independent linear layers of dimensions from 16×16 in SPARKLE256 to 32×32 in SPARKLE512. For these dimensions it may be possible to apply the precise encoding method from [ZR19]. However, due to the large state size, we found it to be still infeasible.

We performed bit-based division property evaluation of the reduced-round SPARKLE permutations. We set $b - 1$ active bits with the inactive bit at index 44 or $44 + b/2$, as offset 44 results in the best bit-based integral characteristic for the ARX-box structure. Furthermore, the branch choice for the inactive bit does not affect the result, due to the rotational branch symmetry (inside each half of the state). The best integral characteristic we found is for 4 steps and an extra Alzette layer, for all three SPARKLE versions. It operates as follows. First, we encrypt the half of the codebook such that one bit in the left half of the input is constant and all other bits are taking all possible values. Then, after 4 steps and the Alzette layer from the 5-th step, the right half of the state sums to zero. We state and prove this characteristic using the structural division property and show that, in fact, fewer active input bits are required. Namely, $64 \cdot n_b + 65$ active bits instead of $2 \cdot 64 \cdot n_b$.

Proposition 4.2.1. *Consider a* SPARKLE*-like permutation of $\mathbb{F}_2^{2h}$, with arbitrary bijective ARX-Boxes permuting $\mathbb{F}_2^m$, arbitrary linear Feistel function and at least 4 branches, i.e. $n_b = 2h/m \geq 4$. Then, the following division property transition is satisfied over 4 steps and an extra ARX-box layer:*

$$\mathcal{D}^{(m,\ldots,m),(m,\ldots,m)}_{(0,\ldots,0,1,m),(m,\ldots,m)} \xrightarrow{A \circ (L \circ A)^4} \mathcal{D}^{(h),(h)}_{(1),(0)} \cup \mathcal{D}^{(h),(h)}_{(0),(2)},$$

[4] For $d \leq 2$, we test the occurrence of all monomials, for $3 \leq d \leq 12$ we choose $N = 10000$ and for $12 \leq d \leq 16$, we choose $N = 1000$.

where A denotes the ARX-box layer and L denotes the linear layer. In other words, the right half of the output sums to zero.

Proof. Without loss of generality, we assume that there is no rotation of branches. Indeed, any permutation of branches inside a half is equivalent to reordering ARX-boxes inside halves and to modifying the Feistel linear layer, which is not constrained in this proposition.

Step 1. The properties $\mathcal{D}_1^m$ and $\mathcal{D}_m^m$ are retained through the ARX-boxes. The right half is fully active, therefore the linear layer does not have mixing effect yet. The following division trail is unique:

$$\mathcal{D}^{(m,\dots,m),(m,\dots,m)}_{(0,\dots,0,1,m),(m,\dots,m)} \xrightarrow{A} \mathcal{D}^{(m,\dots,m),(m,\dots,m)}_{(0,\dots,0,1,m),(m,\dots,m)} \xrightarrow{L} \mathcal{D}^{(m,\dots,m),(m,\dots,m)}_{(m,\dots,m),(0,\dots,0,1,m)}.$$

Step 2. The ARX-box layer does not change anything again. The linear layer allows multiple division trails. Note that at most $(n_b - 1)m - 1$ active bits can be transferred through the Feistel linear function until the right half is saturated to fully active. Therefore, at least $m + 1$ bits remain active in the left half. In particular, at least two branches remain active. As we will show, this is the only requirement to show the proposed balanced property. We reduce active bits to these two in order to cover all possible trails and simplify the proof. Up to permutation of branches inside the state halves,

$$\mathcal{D}^{(m,\dots,m),(m,\dots,m)}_{(m,\dots,m),(0,\dots,0,1,m)} \xrightarrow{A} \mathcal{D}^{(m,\dots,m),(m,\dots,m)}_{(m,\dots,m),(0,\dots,0,1,m)} \xrightarrow{L} \mathcal{D}^{(m,\dots,m),(m,\dots,m)}_{(0,\dots,0),(1,1,0,\dots,0)}.$$

Step 3. The two active branches remain active through the third step, since there is no mixing between them:

$$\mathcal{D}^{(m,\dots,m),(m,\dots,m)}_{(0,\dots,0),(1,1,0,\dots,0)} \xrightarrow{A} \mathcal{D}^{(m,\dots,m),(m,\dots,m)}_{(0,\dots,0),(1,1,0,\dots,0)} \xrightarrow{L} \mathcal{D}^{(m,\dots,m),(m,\dots,m)}_{(1,1,0,\dots,0),(0,\dots,0)}.$$

Step $4 + A$ Similarly, the two active branches stay active after the ARX-box layer of the fourth step:

$$\mathcal{D}^{(m,\dots,m),(m,\dots,m)}_{(1,1,0\dots,0),(0,\dots,0)} \xrightarrow{A} \mathcal{D}^{(m,\dots,m),(m,\dots,m)}_{(1,1,0\dots,0),(0,\dots,0)}.$$

In the linear layer, there are several possibilities. The two active bits from the left half can be transferred to a single branch in the right half by the Feistel function. Then $\mathcal{D}^{(h),(h)}_{(2),(0)}$ is obtained that is mapped through the final ARX-box layer to $\mathcal{D}^{(h),(h)}_{(1),(0)}$, i.e., the left half is possibly not balanced. If one of the active bits is transferred by the linear layer, then $\mathcal{D}^{(h),(h)}_{(1),(1)}$ is obtained, which is covered by the previous case. Otherwise, two active branches remain after the linear layer and after the final ARX-box layer. The output division property in this case is $\mathcal{D}^{(h),(h)}_{(0),(2)}$. The following trails cover all possible trails up to branch permutations in each half:

$$\mathcal{D}^{(m,\dots,m),(m,\dots,m)}_{(1,1,0\dots,0),(0,\dots,0)} \xrightarrow{L} \mathcal{D}^{(m,\dots,m),(m,\dots,m)}_{(2,0,\dots,0),(0,0,\dots,0)}, \xrightarrow{A} \mathcal{D}^{(m,\dots,m),(m,\dots,m)}_{(1,0,\dots,0),(0,0,\dots,0)} \implies \mathcal{D}^{(h),(h)}_{(1),(0)},$$
$$\mathcal{D}^{(m,\dots,m),(m,\dots,m)}_{(1,1,0,\dots,0),(0,\dots,0)} \xrightarrow{L} \mathcal{D}^{(m,\dots,m),(m,\dots,m)}_{(0,\dots,0),(1,1,0,\dots,0)}, \xrightarrow{A} \mathcal{D}^{(m,\dots,m),(m,\dots,m)}_{(0,\dots,0),(1,1,0,\dots,0)} \implies \mathcal{D}^{(h),(h)}_{(0),(2)}.$$

It follows that the following division trail is impossible:

$$\mathcal{D}^{(m,\dots,m),(m,\dots,m)}_{(0,\dots,0,1,m),(m,\dots,m)} \xrightarrow{A\circ(L\circ A)^4} \mathcal{D}^{(h),(h)}_{(0),(1)}$$

Therefore, the right output half is balanced. □

Note that in the proof, a lot of active bits were omitted for simplicity, namely in order to cover all possible trails with a single one. However, as the bit-based division property analysis suggests, a more careful analysis does not yield any longer integral characteristic.

We evaluated also the inverses of the Sparkle permutations. Similarly, the bit-based division property with only one inactive bit (at offset 27 in the left or in the right half) suggested only a general structural distinguisher, similar to the one from Proposition 4.2.1.

Proposition 4.2.2. *Consider a* Sparkle*-like permutation as in Proposition 4.2.1. The following division property transition is satisfied over 4 steps in the reverse direction:*

$$\mathcal{D}^{(m,\dots,m),(m,\dots,m)}_{(m,\dots,m),(0,\dots,0,1,m)} \xrightarrow{(A^{-1}\circ L^{-1})^4} \mathcal{D}^{(h),(h)}_{(2),(0)} \cup \mathcal{D}^{(h),(h)}_{(0),(1)}.$$

Proof. In a similar way to the Proposition 4.2.1, the following division trail covers all division trails:

$$\begin{aligned}
\mathcal{D}^{(m,\dots,m),(m,\dots,m)}_{(m,\dots,m),(0,\dots,0,1,m)} &\xrightarrow{L^{-1}} \mathcal{D}^{(m,\dots,m),(m,\dots,m)}_{(0,\dots,0,1,m),(m,\dots,m)} \xrightarrow{A^{-1}} \mathcal{D}^{(m,\dots,m),(m,\dots,m)}_{(0,\dots,0,1,m),(m,\dots,m)} \\
&\xrightarrow{L^{-1}} \mathcal{D}^{(m,\dots,m),(m,\dots,m)}_{(1,1,0,\dots,0),(0,\dots,0,1,m)} \xrightarrow{A^{-1}} \mathcal{D}^{(m,\dots,m),(m,\dots,m)}_{(1,1,0,\dots,0),(0,\dots,0,1,m)} \\
&\xrightarrow{L^{-1}} \mathcal{D}^{(m,\dots,m),(m,\dots,m)}_{(0,\dots,0),(1,1,0\dots,0)} \xrightarrow{A^{-1}} \mathcal{D}^{(m,\dots,m),(m,\dots,m)}_{(0,\dots,0),(1,1,0\dots,0)},
\end{aligned}$$

And in the last step the same cases take place as in Proposition 4.2.1.

$$\begin{aligned}
\mathcal{D}^{(m,\dots,m),(m,\dots,m)}_{(0,\dots,0),(1,1,0\dots,0)} \xrightarrow{L^{-1}} \mathcal{D}^{(m,\dots,m),(m,\dots,m)}_{(0,\dots,0),(2,0,\dots,0)}, \xrightarrow{A^{-1}} \mathcal{D}^{(m,\dots,m),(m,\dots,m)}_{(0,\dots,0),(1,0,\dots,0)} &\implies \mathcal{D}^{(h),(h)}_{(0),(1)}, \\
\mathcal{D}^{(m,\dots,m),(m,\dots,m)}_{(0,\dots,0),(1,1,0\dots,0)} \xrightarrow{L^{-1}} \mathcal{D}^{(m,\dots,m),(m,\dots,m)}_{(1,1,0,\dots,0),(0,\dots,0)}, \xrightarrow{A^{-1}} \mathcal{D}^{(m,\dots,m),(m,\dots,m)}_{(1,1,0,\dots,0),(0,\dots,0)} &\implies \mathcal{D}^{(h),(h)}_{(2),(0)}.
\end{aligned}$$

The following trail is impossible:

$$\mathcal{D}^{(m,\dots,m),(m,\dots,m)}_{(m,\dots,m),(0,\dots,0,1,m)} \xrightarrow{(A^{-1}\circ L^{-1})^4} \mathcal{D}^{(h),(h)}_{(1),(0)}$$

Therefore, the left output half is balanced. □

4.2.9 Attacks Based on Symmetries

The usage of independent and dense round constants on the *branches* and the addition of different constants before each *step* breaks symmetries within the permutation.

Slide Attacks [BW99]. The XOR of the step counter to the second branch before each step assures that each step is different so that classical slide attacks cannot be applied. As a second source of asymmetry, we XOR the round constant c_i to the right word of the first branch before step i. Similarly to what was done for Keccak, for each Sparkle$n$ version, we further analyzed the function

$$\mathsf{slide}_{n,n_s}(x) = \text{Sparkle}n_{n_s}(x) \oplus \text{Sparkle}n_{n_s+1}(\text{Sparkle}n_1^{-1}(x))$$

with regard to the distribution of low-weight monomials with the same test as described in Section 4.2.8.1. The test passes already after three steps for all Sparkle variants and all $d \leq 16$ (see Table 4.6).

Table 4.6: For each of the slide functions of the Sparkle versions, the table contains the maximum degree d such that the monomial test passes for all $d' \leq d$. We tested up to $d = 16$.

n_s	slide_{256,n_s}	slide_{384,n_s}	slide_{512,n_s}
1	0	0	0
2	0	0	0
3	≥ 16	≥ 16	≥ 16

Rotational Cryptanalysis. Due to the heavy use of dense round constants, we expect Sparkle to be resistant against rotational cryptanalysis. Furthermore, the ℓ function used in the linear layer of Sparkle breaks the rotational symmetry of 32-bit words as well.

4.2.10 Guess and Determine

It might be counter-intuitive because of the long trail approach we use but the diffusion in Sparkle is very fast. At the end of one step, each left branch depends on all the left branches of the input and one branch from the right. Furthermore, not only is the linear Feistel function ensuring this high dependency, it also provides a branching number of 4 (for $n_b \geq 6$), meaning that we cannot find equations linking fewer than 4 branches through the linear layer.

4.3 Attacks Against the Sponge

When absorbing message blocks, we prefer to inject them as branches on the left side of the integral state. There are several reasons for this:

1. these branches are those that will go in the Feistel function the soonest, thus ensuring a quick diffusion of the message blocks in the state,
2. these branches will undergo a double iteration of Alzette right away, meaning that it will be harder for an attacker to control what happens to these branches, and
3. an attacker who wants for instance to find a collision needs to have some control over the branches which receive the blocks injected in the state. By having those be on the left, we ensure that these branches have just received a XOR from the linear Feistel function, and thus that they depend on all the branches that are on the right side at the time of injection. This property makes it harder for an attacker to propagate information backwards or to ensure that some pattern holds right before block injection.

When the rate is higher than a half of the state, we first use all the branches on the left as the inner part and then complete it with as many branches from the right as needed.

4.3.1 Differential Attacks

To study the security of a sponge against differential attacks, we estimate the security parameter for which vanishing differences become unfeasible for an increasing number of steps in the sponge permutation.

First, we observe that the probability $\mathsf{P}_{\mathsf{vanish}}(a)$ of a differential trail covering a absorptions with a sponge with r steps is upper-bounded by U_r^a, where U_r is an upper bound on the probability of all differentials for the r step permutation, unless the difference cancels out at some absorptions. For Sparkle, such bounds are provided in Table 4.3. Let s be the security parameter we aim for. If $(U_r)^2 < 2^{-s}$, then the only way for a vanishing absorbed trail to exist with probability higher than 2^{-s} is for it to correspond to two absorptions, i.e., that the second absorption cancels the difference in the state of the sponge after the absorption of the first difference.

As a consequence, we can restrict our search for absorbed trails to those that have both an input and an output that is fully contained in the outer part of the sponge. The program we used to enumerate all truncated trails to implement a long trail argument is easily modified to only take into account such trails. Then, we upper bound the probability of all the corresponding trail using the same approach as before and we obtain Table 4.7.

Note that while the rate of both Esch256 and Esch384 is equal to 128 bits, it is necessary to look at $n = 384, r = 192$ for Esch256 and $n = 512, r = 256$ for Esch384. Indeed, the indirect injection means that the input and output difference must be over the leftmost 3 and 4 branches respectively. Still, as we can see in Table 4.7, it makes little difference in terms of differential trail probability. Furthermore, it would be necessary for an attacker to find trails that start and end in a specific subspace of the left half of the state.

4.3.2 Linear Attacks

Using a reasoning identical to the one we used above in the differential case, we can restrict ourselves to the case where the input and output masks are restricted to the outer part of the sponge. Doing so, we can look at all the linear trails that would yield linear approximations connecting the outer

Table 4.7: The quantity $-\log_2(p)$ where p is an upper bound on the probability of a differential trail over one call to Sparkle where both the input and the output differences are fully contained in the outer part. The $\emptyset$ symbols means that such trails impossible. $r \to r$ denotes "outer part to outer part" trails and $r \to n$ denotes trails where the input is in the outer part but the output is not constrained.

n	r	c (security)	Type	3	4	5	6	7	8
256			$n \to n$	64	88	140	168	192	216
	192	64	$r \to r$	76	108	140	168	204	232
			$r \to n$	64	108	140	168	192	232
	128	128	$r \to r$	96	128	192	192	224	≥ 256
			$r \to n$	96	116	140	172	212	244
	64	192	$r \to r$	$\emptyset$	128	192	192	≥ 256	≥ 256
			$r \to n$	96	128	148	172	212	≥ 256
384			$n \to n$	70	100	178	200	230	260
	256	128	$r \to r$	108	148	180	200	268	296
			$r \to n$	70	140	178	200	230	296
	192	192	$r \to r$	128	160	256	256	288	320
			$r \to n$	128	148	178	210	276	306
	128	256	$r \to r$	128	160	256	256	320	320
			$r \to n$	128	160	180	210	276	306
	64	320	$r \to r$	$\emptyset$	160	256	256	320	320
			$r \to n$	128	160	180	210	276	306
512			$n \to n$	76	112	210	232	268	276
	256	256	$r \to r$	160	192	256	320	352	416
			$r \to n$	134	172	212	248	332	372
	128	384	$r \to r$	$\emptyset$	192	256	320	352	384
			$r \to n$	134	172	212	248	332	376
	64	448	$r \to r$	$\emptyset$	192	320	320	384	384
			$r \to n$	160	192	212	248	340	376

parts of the internal state before and after a call to the Sparkle permutation. If this absolute correlation is too high, it could lead for instance to observable biases in a keystream generated using a Schwaemm instance. Table 4.8 bounds such probabilities.

4.3.3 Impossible Differentials

The 4-step impossible differential described above in Section 4.2.6 cannot be applied when the permutation is used in a sponge unless the rate is larger than the capacity (as it is in Schwaemm256-128). Therefore, for Sparkle384, the four-step impossible differential given above, i.e.,

$$\begin{array}{lccccccc} \text{input}: & 0 & 0 & 0 & \alpha & 0 & 0 & \downarrow \\ \text{step 1}: & 0 & 0 & \gamma & 0 & 0 & 0 & \downarrow \\ \text{step 2}: & & & & \not{\downarrow} & & & \\ \text{step 3}: & 0 & 0 & 0 & \delta & 0 & 0 & \uparrow \\ \text{step 4}: & 0 & 0 & \beta & 0 & 0 & 0 & \uparrow \end{array},$$

Table 4.8: The quantity $-\log_2(p)$ where p is an upper bound on the absolute correlation of a linear trail connecting the outer part of the input with the outer part of the output of various SPARKLE instances. The $\emptyset$ symbols means that the trails connecting the outer part to itself are impossible.

n	r	c (security)	3	4	5	6	7
256	192	64	23	42	57	76	91
	128	128	23	55	74	76	91
	64	192	59	55	89	89	123
384	256	128	25	46	76	97	110
	192	192	25	72	93	97	110
	128	256	42	72	108	110	142
	64	320	$\emptyset$	72	123	123	157
512	256	256	27	78	97	129	129
	128	384	$\emptyset$	89	110	142	161

is valid in the two settings of SCHWAEMM256-128. However, in the setting of ESCH256 and SCHWAEMM192-192, we only have a three-round impossible differential

$$\begin{array}{lccccccc} \text{input}: & 0 & \alpha & 0 & 0 & 0 & 0 & \downarrow \\ \text{step 1}: & 0 & 0 & 0 & 0 & \gamma & 0 & \downarrow \\ \text{step 2}: & & & & \lightning & & & \\ \text{step 3}: & 0 & \beta & 0 & 0 & 0 & 0 & \uparrow \end{array}.$$

For SPARKLE256 in our sponge settings, there is only a three-step impossible differential given as

$$\begin{array}{lcccccc} \text{input}: & 0 & \alpha & 0 & 0 & \downarrow \\ \text{step 1}: & 0 & 0 & 0 & \gamma & \downarrow \\ \text{step 2}: & & \lightning & & & \\ \text{step 3}: & 0 & \beta & 0 & 0 & \uparrow \end{array}.$$

Similarly, for SPARKLE512 in our sponge setting, there is only a three-step impossible differential given by

$$\begin{array}{lccccccccc} \text{input}: & 0 & 0 & 0 & \alpha & 0 & 0 & 0 & 0 & \downarrow \\ \text{step 1}: & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \gamma & \downarrow \\ \text{step 2}: & & & & \lightning & & & & & \\ \text{step 3}: & 0 & 0 & 0 & \beta & 0 & 0 & 0 & 0 & \uparrow \end{array}.$$

4.3.4 Zero-correlation Attacks

The 4-step zero-correlation distinguisher described above cannot be applied when the permutation is used in a sponge unless the rate is larger than the capacity (as it is in SCHWAEMM256-128). Therefore, for SPARKLE384, the four-step zero-correlation distinguisher given above, i.e.,

$$\begin{array}{lccccccc} \text{input}: & 0 & 0 & \alpha & 0 & 0 & 0 & \downarrow \\ \text{step 1}: & 0 & 0 & 0 & 0 & 0 & 0 & \downarrow \\ \text{step 2}: & & & & \lightning & & & \\ \text{step 3}: & 0 & 0 & 0 & 0 & 0 & 0 & \uparrow \\ \text{step 4}: & 0 & 0 & 0 & \beta & 0 & 0 & \uparrow \end{array},$$

is valid in the setting of SCHWAEMM256-128. However, in the two settings of ESCH256 and SCHWAEMM192-192, we only have a three-round zero-correlation distinguisher

$$\begin{array}{lccccccc} \text{input}: & 0 & \alpha & 0 & 0 & 0 & 0 & \downarrow \\ \text{step 1}: & 0 & 0 & 0 & 0 & 0 & 0 & \downarrow \\ \text{step 2}: & & & & \not{\downarrow} & & & \\ \text{step 3}: & 0 & \beta & 0 & 0 & 0 & 0 & \uparrow \end{array}.$$

For SPARKLE256 in our sponge settings, there is only a three-step zero-correlation distinguisher given as

$$\begin{array}{lcccc} \text{input}: & 0 & \alpha & 0 & 0 & \downarrow \\ \text{step 1}: & 0 & 0 & 0 & 0 & \downarrow \\ \text{step 2}: & & \not{\downarrow} & & & \\ \text{step 3}: & 0 & \beta & 0 & 0 & \uparrow \end{array}.$$

Similarly, for SPARKLE512 in our sponge setting, there is only a three-step zero-correlation distinguisher given by

$$\begin{array}{lcccccccc} \text{input}: & 0 & 0 & 0 & \alpha & 0 & 0 & 0 & 0 & \downarrow \\ \text{step 1}: & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \downarrow \\ \text{step 2}: & & & & \not{\downarrow} & & & & & \\ \text{step 3}: & 0 & 0 & 0 & \beta & 0 & 0 & 0 & 0 & \uparrow \end{array}.$$

4.4 Guess and Determine

As argued in Section 4.2.10, diffusion is fast in SPARKLE. This section shows several attacks on round-reduced variants of SCHWAEMM. The attacks are summarized in Table 4.9.

Table 4.9: Guess and determine attacks on SCHWAEMM instances. ϵ is an arbitrary positive parameter. 0.5 step denotes an extra Alzette layer.

Instance	Steps	Whitening	Method	Time	Data
SCHWAEMM128-128	3.5	no	data trade-off	2^{64}	2^{64}
SCHWAEMM192-192	3.5	no	data trade-off	2^{128}	2^{64}
SCHWAEMM256-256	3.5	no	data trade-off	2^{192}	2^{64}
SCHWAEMM256-256	3.5	no	guess and det.	2^{192}	1
SCHWAEMM128-128	4.5	no	birthday diff.	$2^{96+\epsilon}$	$2^{96-\epsilon}$
SCHWAEMM192-192	4.5	no	birthday diff.	$2^{128+\epsilon}$	$2^{128-\epsilon}$
SCHWAEMM256-256	4.5	no	birthday diff.	$2^{192}+2^{160+\epsilon}$	$2^{160-\epsilon}$
SCHWAEMM256-256	3.5	yes	birthday diff.	$2^{224+\epsilon}$	$2^{224-\epsilon}$

4.4.1 Notation used in the Attacks

Consider an instance of SCHWAEMM. Let $\mathbf{A}_j^i$ denote j-th instance of Alzette at the left half of the state at step i together with the step constant addition:

$$\mathbf{A}_j^i(x) = \begin{cases} A_{c_0}(x \oplus c_i), & \text{if } j = 0, \\ A_{c_1}(x \oplus i), & \text{if } j = 1, \\ A_{c_j}(x), & \text{if } 2 \le j < h_b. \end{cases}$$

Let $\mathbf{B}_j^i$ denote the j-th instance of Alzette at the right half of the state at step i: $\mathbf{B}_j^i = A_{c_{h_b+j}}$. Let $\mathbf{A}^i$ denote the parallel application of $\mathbf{A}_0^i, \ldots, \mathbf{A}_{h_b-1}^i$; $\mathbf{B}^i$ denote the parallel application of $\mathbf{B}_0^i, \ldots, \mathbf{B}_{h_b-1}^i$.

Let $\mathbf{X}[a]$ denote the map $x \mapsto (x \oplus a)$. Let $\mathbf{M}$ denote the linear Feistel map $\mathcal{M}_{h_b}$ and let ℓ' denote the linear feed-forward function used in $\mathbf{M}$:

$$\ell'((x_1||x_2),(y_1||y_2)) = (y_2||y_1 \oplus y_2),(x_2||x_1 \oplus x_2), \quad \text{where } x_1, x_2, y_1, y_2 \in \mathbb{F}_2^{16}.$$

Let $\mathbf{R}$ denote the rotation of h_b branches to the left by one position:

$$\mathbf{R}(x_0, \ldots, x_{h_b-1}) = (x_1, \ldots, x_{h_b-1}, x_0).$$

Consider a known-plaintext scenario. The outer part of the state becomes known before and after a call to a (round-reduced) SPARKLE permutation. Let m_{in} be the initial outer part and m_{out} be the final outer part. We call the Alzette layer a *half-step*. Note that in the considered scenario, any attack on t full steps can be trivially extended to $t + 1/2$ steps, since the final calls to Alzette in the outer part can be easily inverted.

4.4.2 Differential Assumptions on the Alzette Instances

A single isolated iteration of Alzette does not have a strong resistance against differential attacks. Indeed, there is a differential trail with probability 2^{-6}. In our attacks, we assume that particular problems about differential transitions involving random differences can be solved efficiently, even though we do not propose concrete algorithms. For example, consider the problem of checking whether a random differential transition over an ARX-box is possible. A naive approach would require 2^{64} evaluations. However, we can expect that with a meet-in-the-middle method it can be done much more efficiently. Indeed, Alzette has only 4 rounds. We further assume that the difference distribution table (DDT) of an ARX-box is very sparse and such problems about differential transitions have few solutions on average.

The problems we consider are about finding all solutions of the following differential transition types:

Problem 1. $a \xrightarrow{A} b$, where $a, b \in \mathbb{F}_2^{64}$ are known random differences, A is an ARX-box or the inverse of an ARX-box,

Problem 2. $a \xrightarrow{A} \alpha, b \xrightarrow{B} \alpha$, where $a, b \in \mathbb{F}_2^{64}$ are known random differences, A, B are ARX-boxes or their inverses, $\alpha \in \mathbb{F}_2^{64}$ is an unknown difference.

Problem 3. $\alpha \xrightarrow{A} \beta, \alpha \xrightarrow{B} \beta + a$, where $a \in \mathbb{F}_2^{64}$ is a known random difference, A, B are ARX-boxes or their inverses, $\alpha, \beta \in \mathbb{F}_2^{64}$ are unknown differences.

We denote the average ratio of solutions to a problem by ν, and the average time to enumerate all solutions by τ_f.

4.4.3 Birthday-Differential Attacks

Encryptions with unique nonces can be expected to be completely independent. Therefore, a nonce-respecting adversary can not easily inject differences in the state in the encryption queries. Indeed, the difference between two encryptions in any part of the state can be expected to be random, and independent of the message due to the state randomization by the initialization with unique nonces. However, any fixed difference in n-bit part of the state may be obtained randomly among approximately $2^{n/2}$ random states. Therefore, with $2^{n/2}$ data, we can expect to have a pair satisfying an n-bit differential constraint. However, the procedure of finding this pair in the pool of encryptions has to be efficient.

The most useful differentials for this attack method are zero differences on full branches. They propagate to zero difference through Alzette. It is also desirable that this differences imply the zero difference of some function of observable parts of the state (i.e. m_{in}, m_{out}). Then a hash table can be used to filter pairs from the data pool efficiently.

Proposition 4.4.1. *Assume that 64η bits in the encryption process are chosen such that for a pair of encryptions having zero difference in those 64η bits,*

1. *64μ bits can be efficiently computed from m_{in}, m_{out} (denote the function by π), such that they also have zero difference;*
2. *pairs of encryptions that satisfying the zero difference can be further filtered in time τ_f, keeping a fraction of most $\nu^{\eta-\mu}$ pairs (denote the function by* filter

3. given such a pair, the full state can be recovered in time τ_r (denote the function by recover*).*

Then, the full state can be recovered using $2^{64\eta/2+1/2}$ data and $2^{64(\eta-\mu)}(\tau_f + \nu^{\eta-\mu}\tau_r)$ time. The general attack procedure is given in Algorithm 4.1.

Proof. There are $2^{64\eta}$ pairs in the encryption pool and we can expect to have a pair having the required zero difference with a high probability. The complexity of the initial filtering by π can be neglected. Therefore, we assume that all $2^{64(\eta-\mu)}$ pair candidates (on average) can be enumerated efficiently. For each candidate, the verification and, in case of verification success, the state recovery take time $\tau_f + \nu^t\tau_r$. □

Algorithm 4.1 Birthday-Differential attack procedure.

collect $2^{64\eta/2+1/2}$ known-plaintext encryptions
compute corresponding outer parts m_{in}, m_{out}
store $\pi(m_{in}, m_{out})$ for each encryption in a hash table
for all $(m_{in}, m_{out}), (m'_{in}, m'_{out})$ such that $\pi(m_{in}, m_{out}) = \pi(m'_{in}, m'_{out})$ **do**
  if filter$((m_{in}, m_{out}), (m'_{in}, m'_{out}))$ **then**
    $s \leftarrow$ recover$((m_{in}, m_{out}), (m'_{in}, m'_{out}))$
    return s
  end if
end for

Attacks of this type typically have quite large data complexity, violating the data limit set in the specification. However, it should be noted, that the actual key used does not matter as each state is always expected to be random and independent. Therefore, re-keying does not prevent the attack. If the required difference is achieved by a pair of encryptions under different keys, then both states are recovered by the attack.

An adversary can further exploit this fact. The data complexity may be reduced by performing a precomputation. The adversary encrypts 2^{64t} data (t may be fractional), and forms a pool in the same way as in the normal attack. Then, $2^{64(\eta-t)}$ data is collected from encryptions under the unknown secret key. Among the too pools, there are $2^{64\eta}$ pairs and at least one pair is expected to satisfy the zero difference with a high probability. Note that the data reduction starts only with $t > \eta/2$ and is costly in the time and memory complexity.

4.4.3.1 Attack on 3.5-step Schwaemm Instances without Rate Whitening

Consider an instance of Schwaemm with the rate equal to the capacity (i.e. one of Schwaemm128-128, Schwaemm192-192, Schwaemm256-256), which uses the Sparkle permutation reduced to 3 steps and has no rate whitening.

Let y denote the output of the linear Feistel function $\mathbf{M}$ in second step (as shown in Figure 4.2). It lies on the following *cyclic* structure (marked with dashed red rectangle):

$$y = \mathbf{M} \circ (\mathbf{B}^2)^{-1} \circ \mathbf{X}\left[\mathbf{R}^{-1}(m_{out})\right] \circ \mathbf{M} \circ \mathbf{A}^2 \circ \mathbf{R} \circ \mathbf{X}\left[\mathbf{B}^1(\mathbf{A}^0(m_{in}))\right] \circ (y).$$

Let

$$m'_{in} = \mathbf{B}^1(\mathbf{A}^0(m_{in})),$$
$$m'_{out} = \mathbf{M}^{-1}(\mathbf{R}^{-1}(m_{out})).$$

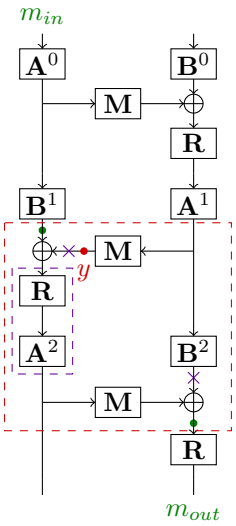
Then

$$y = \left(\mathbf{M} \circ (\mathbf{B}^2)^{-1} \circ \mathbf{M}\right) \circ \left(\mathbf{X}\left[m'_{out}\right] \circ \mathbf{A}^2 \circ \mathbf{R} \circ \mathbf{X}\left[m'_{in}\right]\right)(y), \text{ and} \tag{4.2}$$

$$\mathbf{M}^{-1} \circ \mathbf{B}^2 \circ \mathbf{M}^{-1}(y) = \mathbf{X}\left[m'_{out}\right] \circ \mathbf{A}^2 \circ \mathbf{R} \circ \mathbf{X}\left[m'_{in}\right](y). \tag{4.3}$$

Note that Equation 4.2 shows that the unknown part of the state y is a *fixed point* of a particular bijective structure using the constants m'_{in}, m'_{out}. This is an interesting formulation of the constraint on the unknown part of the state.

Figure 4.2: Attack on 3.5-step SPARKLE without whitening. The green dots show known values, the purple crosses show zero differences in the birthday-differential attack. The red dashed area highlights the part being attacked, the purple dashed area shows the part with the target differential transition in the birthday-differential attack.



Precomputation/Data Trade-off Attack. Note that the left part of Equation 4.3 is independent of m'_{in}, m'_{out}. Moreover, the right part consists of independent ARX-boxes. Therefore, guessing one 64-bit branch of y leads to knowledge of an input and an output 64-bit branches of the function from the left-hand side. A data trade-off attack follows. The trade-off parameterized by an integer r, $0 < r \leq 64$.

We start by the precomputation phase. Let $z = \mathbf{M}^{-1} \circ \mathbf{B}^2 \circ \mathbf{M}^{-1}(y)$. We iterate over all $y_1 \in \mathbb{F}_2^{64-r}$ and all $y_i \in \mathbb{F}_2^{64}$ for $i \neq 1$, and generate the table mapping (y_1, z_0) to all values y satisfying the constraint. On average, we expect $2^{64h_b-r}/2^{64} = 2^{64(h_b-1)-r}$ candidates per each (y_1, z_0) in the table. This step requires 2^{64h_b-r} time and memory blocks.

In the online phase, we collect 2^r known plaintexts-ciphertext pairs and compute the corresponding m'_{in}, m'_{out} for each pair. Then, for each $y_1 \in \mathbb{F}_2^{64-r}$ we compute

$$z_0 = (m'_{out})_0 \oplus \mathbf{A}_0^2((m'_{in})_1 \oplus y_1).$$

For each preimage candidate of (y_1, z_0) in the precomputed table, we recover the full state in the middle of the second step. We then check if the corresponding state correctly connects m_{in}, m_{out} and possibly recover the secret key by inverting the sponge operation.

If a considered plaintext-ciphertext pair is such that the leftmost r bits of y_1 are equal to zero, then the attack succeeds. Indeed, then, for one of the guesses of y_1, the pair (y_1, z_0) corresponds to the correct preimage. For each of 2^r plaintext-ciphertext pairs we guess 2^{64-r} values of (y_1, z_0). Correct y_1 identifies a table mapping the z_0 to all possible y. Therefore, on average, there will be $2^{64-r} \cdot 2^{64(h_b-1)-r} = 2^{64h_b-2r}$ total candidates. The time required to check a candidate and to recover the secret key is negligible.

The online phase requires 2^r different 2-block plaintext-ciphertext pairs, 2^{64h_b-2r} time and negligible amount of extra memory.

The following attacks on SCHWAEMM instances follow:

1. SCHWAEMM128-128: with $r = 64$, the full attack requires 2^{64} time, memory and data; with $r = 32$, the full attack requires 2^{96} time and memory, and 2^{32} data.
2. SCHWAEMM192-192: with $r = 64$, the full attack requires 2^{128} time, memory and 2^{64} data.
3. SCHWAEMM256-256: with $r = 64$, the full attack requires 2^{192} time, memory and 2^{64} data.

Low-data Variant of the Attack on Schwaemm256-256. Due to the high branching number of $\mathbf{M}$, it is hard to exploit the structure of the function $\mathbf{M}^{-1} \circ \mathbf{B}^2 \circ \mathbf{M}^{-1}$ by guessing several branches. However, for the largest instance Schwaemm256-256, a simple attack requiring one known-plaintext and 2^{192} time is possible.

The key observation is that when $\ell'(x)$ is fixed, $\mathbf{M}(x)$ splits into h_b independent xors with $\ell'(x)$. In the attack, we simply guess the corresponding ℓ' for the two calls to $\mathbf{M}$. Precisely, let $\ell'_y = \ell'(\mathbf{M}^{-1}(y))$ and $\ell'_z = \ell'(\mathbf{M}^{-1} \circ \mathbf{B}^2 \circ \mathbf{M}^{-1}(y))$. The computations from Equation 4.2 then split into one large cycle:

$$\begin{aligned}
y_0 &= \mathbf{X}\left[\ell'_y\right] \circ (\mathbf{B}_0^2)^{-1} \circ \mathbf{X}\left[\ell'_z\right] \circ \mathbf{X}\left[(m'_{out})_0\right] \circ \mathbf{A}_0^2 \circ \mathbf{X}\left[(m'_{in})_1\right](y_1),\\
y_1 &= \mathbf{X}\left[\ell'_y\right] \circ (\mathbf{B}_1^2)^{-1} \circ \mathbf{X}\left[\ell'_z\right] \circ \mathbf{X}\left[(m'_{out})_1\right] \circ \mathbf{A}_1^2 \circ \mathbf{X}\left[(m'_{in})_2\right](y_2),\\
y_2 &= \mathbf{X}\left[\ell'_y\right] \circ (\mathbf{B}_2^2)^{-1} \circ \mathbf{X}\left[\ell'_z\right] \circ \mathbf{X}\left[(m'_{out})_2\right] \circ \mathbf{A}_2^2 \circ \mathbf{X}\left[(m'_{in})_3\right](y_3),\\
y_3 &= \mathbf{X}\left[\ell'_y\right] \circ (\mathbf{B}_3^2)^{-1} \circ \mathbf{X}\left[\ell'_z\right] \circ \mathbf{X}\left[(m'_{out})_3\right] \circ \mathbf{A}_3^2 \circ \mathbf{X}\left[(m'_{in})_0\right](y_0).
\end{aligned}$$

Let us guess y_0 and compute the whole cycle. If the result matches guessed y_0, then we obtain a candidate for the full $y = (y_0, y_1, y_2, y_3)$. On average, we can expect to find one false-positive candidate.

The attack requires 1 known plaintext-ciphertext pair, negligible amount of memory, and 2^{192} time.

Birthday-Differential Attack. A birthday-differential attack can be mounted too. We are looking for a pair having zero difference in y. Then the expression

$$\mathbf{X}\left[m'_{out}\right] \circ \mathbf{A}^2 \circ \mathbf{R} \circ \mathbf{X}\left[m'_{in}\right](y) \tag{4.4}$$

has zero difference in the input y and zero difference in the output. Therefore, the difference in m'_{in} is transformed into the difference in $\mathbf{R}(m'_{out})$ by an Alzette layer. This is the first problem we noted in Section 4.4.2.

Note that the amount of pairs of encryptions in the pool has to be greater than 2^{64h_b} in order for a pair with zero difference in y to exist. Therefore, enumeration of all pairs and checking the possibility of the differential transition $m'_{in} \xrightarrow{\mathbf{A}^2 \circ \mathbf{R}} m'_{out}$ results in an ineffective attack.

As described in the birthday-differential attack framework, we further strengthen the constraints in order to obtain an efficient initial filtering. We require that t branches of m'_{in} starting from the second branch have zero difference too, $0 < t < h_b$. Then, m'_{out} must have zero difference in the first t branches. This allows us to obtain initial filtering with $\mu = 2t$, i.e. with the probability $2^{-64 \cdot 2t}$. In total we need zero difference in $h_b + t$ branches. Therefore, we need $2^{64(h_b+t)/2+1/2}$ data and we expect to keep $2^{64(h_b+t)} \cdot 2^{-64 \cdot 2t} = 2^{64(h_b-t)}$ pairs on average after the initial filtering procedure.

The second filtering step is based on filtering possible differential transitions. In the correct pair, the differences $\Delta m'_{in}$ and $\Delta m'_{out}$ of the values m'_{in} and m'_{out} respectively are related by the layer $\mathbf{A}^2$ of Alzette calls. More precisely, for all $i, 0 \le i < h_b$, the following differential transition holds:

$$(\Delta m'_{in})_{i+1} \xrightarrow{\mathbf{A}_i^2} (\Delta m'_{out})_i.$$

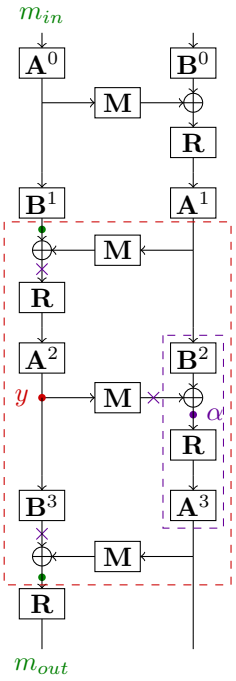
Verifying a pair requires checking whether a differential transition over an Alzette instance is possible or not (see Problem 1 in Section 4.4.2). We assume that only a fraction ν of all differential transitions over an Alzette instance is possible, and that for any differential transitions all solutions can be found in time τ_f on average.

The branch values corresponding to zero difference transitions can be found exhaustively in time $\tau_r \le 2^{64t}$ or more efficiently by exploiting the structure further.

We estimate the final complexity of the attack by $2^{64(h_b+t)/2+1/2}$ data and $2^{64(h_b-t)}(\tau_f + \nu^{h_b-t}\tau_r)$ time. Assuming low values of ν, τ_f and τ_r, we estimate the following attack complexities for different instances of Schwaemm:

1. Schwaemm128-128: with $t = 1$, the attack requires $2^{96.5}$ data, and slightly more than 2^{64} time. By the precomputation cost of $2^{96+\epsilon}$ time and memory, the data requirement can be reduced to $2^{96-\epsilon}$ for any $\epsilon < 32$.

Figure 4.3: Attack on 4.5-step SPARKLE without whitening. The green dots show known values, the purple crosses show zero differences. The red dashed area highlights the part being attacked, the purple dashed area shows the part with the target differential transition.



2. SCHWAEMM192-192: with $t = 1$, the attack requires $2^{128.5}$ data, and slightly more than 2^{128} time. By the precomputation cost of $2^{128+\epsilon}$ time and memory, the data requirement can be reduced to $2^{128-\epsilon}$ for any $\epsilon < 64$.

3. SCHWAEMM256-256: with $t = 1$, the attack requires $2^{160.5}$ data, and slightly more than 2^{192} time. By the precomputation cost of $2^{160+\epsilon}$ time and memory, the data requirement can be reduced to $2^{160-\epsilon}$ for any $\epsilon < 96$.

4.4.3.2 Attack on 4.5-step Schwaemm without Rate Whitening

Consider an instance of SCHWAEMM with the rate equal to the capacity (i.e. one of SCHWAEMM128-128, SCHWAEMM192-192, SCHWAEMM256-256), which uses the SPARKLE permutation reduced to 4 steps and has no rate whitening.

Let y be the input to the linear Feistel layer $\mathbf{M}$ in the third step (see Figure 4.3). We aim to mount a birthday-differential attack with zero-difference in y. The parts of the structure with zero difference are marked with purple crosses in the figure. It follows that differences of the observed rate parts can be propagated and connected by independent branches. More formally, let

$$m'_{in} = \mathbf{M}^{-1}(\mathbf{B}^1(\mathbf{A}^0(m_{in}))),$$
$$m'_{out} = \mathbf{M}^{-1}(\mathbf{R}(m_{out})).$$

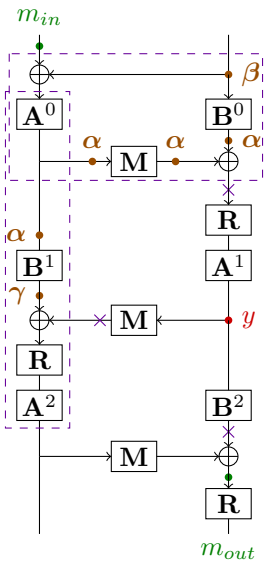
Denote the difference in m'_{in} by $\Delta m'_{in}$, and the difference in m'_{out} by $\Delta m'_{out}$. It follows that the difference $\Delta m'_{in}$ propagates through $\mathbf{B}^2$ into the same difference as the difference $\Delta m'_{out}$ propagates through $\mathbf{R}^{-1} \circ (\mathbf{A}^3)^{-1}$. Note that they are connected by h_b independent 64-bit branches:

$$(\Delta m'_{in})_i \xrightarrow{\mathbf{B}_i^2} \alpha_i \xleftarrow{(\mathbf{A}_{i-1}^3)^{-1}} (\Delta m'_{out})_{i-1}, \tag{4.5}$$

where α is the unknown intermediate difference.

In order to make the birthday-differential attack, we further strengthen the zero-difference constraint in order to perform an efficient initial filtering. We require that $(\Delta m'_{in})_i = \alpha_i =$

Figure 4.4: Attack on 3.5-step SPARKLE with whitening. The green dots show known values, the purple crosses show zero differences. The purple dashed areas shows the parts with the target differential transitions.



$(\Delta m'_{out})_{i-1} = 0$ for all $i < t$ for an integer t, $0 < t < h_b$. This constraints allows us to filter the pairs efficiently by the zero-difference parts of m'_{in} and m'_{out}.

The second filtering step is based on checking the possibility of the differential transitions from Equation 4.5. This is Problem 2 mentioned in Section 4.4.2.

Similarly to the previous attack, the final complexity of the attack is estimated by $2^{64(h_b+t)/2+1/2}$ data and $2^{64(h_b-t)}(\tau_f + \nu^{h_b-t}\tau_r)$ time. Under the assumption of low values of ν, τ_f and τ_r, the following attacks are derived:

1. SCHWAEMM128-128: with $t = 1$, the attack requires $2^{96.5}$ data, and more than 2^{64} time. By the precomputation cost of $2^{96+\epsilon}$ time and memory, the data requirement can be reduced to $2^{96-\epsilon}$ for any $\epsilon < 32$.

2. SCHWAEMM192-192: with $t = 1$, the attack requires $2^{128.5}$ data, and more than 2^{128} time. By the precomputation cost of $2^{128+\epsilon}$ time and memory, the data requirement can be reduced to $2^{128-\epsilon}$ for any $\epsilon < 64$.

3. SCHWAEMM256-256: with $t = 1$, the attack requires $2^{160.5}$ data, and more than 2^{192} time. By the precomputation cost of $2^{160+\epsilon}$ time and memory, the data requirement can be reduced to $2^{160-\epsilon}$ for any $\epsilon < 96$.

4.4.3.3 Attack on 3.5-step Schwaemm256-256

Consider SCHWAEMM256-256, which uses the SPARKLE permutation reduced to 3 steps and has the rate whitening.

Let y be the input to the linear Feistel function $\mathbf{M}$ in the second step (see Figure 4.4). We aim to find a pair of encryptions with zero difference in y. We further restrict the input and the output difference of $\mathbf{M}$ in the first round to have form $\boldsymbol{\alpha} = (\alpha, \alpha, 0, 0)$ for any $\alpha \in \mathbb{F}_2^{64}$. Note that this happens in the fraction $2^{-3 \cdot 64}$ of all inputs to M, because $(\alpha, \alpha, 0, 0)$ is always mapped to $(\alpha, \alpha, 0, 0)$ by M. In total, we require 7 independent branches to have zero difference.

First, observe that for some $\boldsymbol{\beta} \in (\mathbb{F}_2^{64})^4 = (\beta_0, \beta_1, 0, 0)$, the following differential transitions hold (see the topmost purple area in Figure 4.4):

$$\boldsymbol{\alpha} \xrightarrow{\mathbf{A}^0} \beta \oplus \Delta m_{in},$$
$$\boldsymbol{\alpha} \xrightarrow{\mathbf{B}^0} \beta,$$

where Δm_{in} is the difference in m_{in}. It follows that $(\Delta m_{in})_2 = (\Delta m_{in})_3 = 0$, because $\boldsymbol{\alpha}_2 = \boldsymbol{\alpha}_3 = 0$. For $i = 0$ and $i = 1$ we obtain an instance of Problem 3 from Section 4.4.2:

$$\alpha \xrightarrow{\mathbf{A}_i^0} \beta_i \oplus (\Delta m_{in})_i,$$
$$\alpha \xrightarrow{\mathbf{B}_i^0} \beta_i.$$

Note that here the same unknown $\alpha \in \mathbb{F}_2^{64}$ appears in two instances of the problem, thus adding more constraints on α. Consider the leftmost purple area in Figure 4.4. It describes another differential transition:

$$\boldsymbol{\alpha} \xrightarrow{\mathbf{B}^1} \boldsymbol{\gamma} \xrightarrow{\mathbf{A}^2 \circ \mathbf{R}} \Delta m'_{out},$$

where $\boldsymbol{\gamma} \in (\mathbb{F}_2^{64})^4 = (\gamma_0, \gamma_1, 0, 0)$ and $\Delta m'_{out}$ is the difference of $m'_{out} = \mathbf{M}^{-1}(\mathbf{R}^{-1}(m_{out}))$. It follows that $(\Delta m'_{out})_1 = (\Delta m'_{out})_2 = 0$ and for $i = 0$ and $i = 1$, the following differential transition holds:

$$\alpha \xrightarrow{\mathbf{B}_i^1} \boldsymbol{\gamma}_i \xrightarrow{\mathbf{A}_{i-1}^2} (\Delta m'_{out})_{i-1}.$$

In total, $\eta = 7$ branches are constrained to have zero difference and $\mu = 4$ branches with zero differences can be observed from m_{in}, m_{out}, providing strong initial filter. Using $2^{64\eta/2+1/2}$ data, we expect to get $2^{64(\eta-\mu)} = 2^{64\cdot 3}$ encryption pairs after the initial filtering. Furthermore, we assume that the constraints on the unknown difference $\alpha \in \mathbb{F}_2^{64}$ are very strong and are enough to significantly reduce the number of possible encryption pairs. We assume it can be done efficiently, since a precomputation time of $2^{64\cdot 3}$ is available. After values of branches involved in differential transitions with α are recovered, the rest of the state can be recovered in negligible time.

Therefore, we estimate the data complexity of the attack by $2^{224.5}$ and same time complexity (the heavy filtering step has to filter 2^{192} pairs). By precomputations costing $2^{224+\epsilon}$ time and memory, the data complexity may be reduced to $2^{224-\epsilon}$, for any $\epsilon < 32$.

This attack does not directly apply to Schwaemm128-128, Schwaemm192-192 since the constraint on the linear map $\mathbf{M}$ in the first step is too costly. For Schwaemm192-192 with $\boldsymbol{\alpha} = (\alpha, \alpha, 0)$ we would obtain $\eta = 5, \mu = 2$ leaving with 2^{192} pair candidates, which is too much to filter in time 2^{192}. Therefore, a stronger initial filter is required.

5 Implementation Aspects

5.1 Software Implementations

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

5.1.1 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.

5.1.2 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 5.1. It is the strategy we used in Appendix A. 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 5.1 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.

5.1.3 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.

5.2 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 Kogge–Stone 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.

5.3 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 mul-

tiplicative (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. [CGTV15, 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.

5.4 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 2.1. 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 5.1. 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 5.2). All execution times for ARM specified in Table 5.1 were obtained with the cycle-accurate instruction set simulator of Keil MicroVision using a generic Cortex-M3 model as target

Table 5.1: 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.

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)

device.[1] 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).

Table 5.2: 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.

Permutation	Code Size (byte)	Stack Usage (byte)
SPARKLE256	316+32	40
SPARKLE384	452+32	48
SPARKLE512	596+32	52

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) assembler implementations of the underlying permutation for the Cortex-M series of ARM microcontrollers. Table 5.3 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.[2] 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

[1] 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 5.1 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 [BKL+17a].

[2] 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.

Table 5.3: Comparison of fully unrolled ARMv7-M Assembler implementations of the permutations of Ascon, Sparkle384, Gimli and Xoodoo on a Cortex-M3 microcontroller.

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

6-fold increase of code size. Also given in Table 5.3 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.

Table 5.4: 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.

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 5.4 shows the AVR execution times and throughputs of the different 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 that uses the hand-written assembler code for the Sparkle permutation. Table 5.5 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 5.6 summarizes the implementation results of Esch256, SHA-2, SHA-3, some SHA-3 finalists, as well as Gimli [BKL+17a]. 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 [BEE+13] were obtained with "pure" assembler implementations, whereas Esch256 contains hand-optimized as-

Table 5.5: 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.

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)

sembler 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.

Table 5.6: 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.

Hash function	Ref.	Throughput (c/b)	Code size (b)
Esch256	This paper	578	1428
Blake-256	[BEE+13]	562	1166
Gimli-Hash small	[BKL+17b]	1610	778*
Gimli-Hash fast	[BKL+17b]	725	19218*
Groestl-256	[BEE+13]	686	1400
JH-256	[BEE+13]	5062	1020
Keccak†	[BEE+13]	1432	868
SHA-256	[BEE+13]	532	1090

* The code size corresponds to the permutation alone.

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

A comparison of the performance of hash functions is easily possible because there exist a number of implementation results in the literature (e.g. [BEE+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.

Bibliography

[AC17] Carlisle Adams and Jan Camenisch, editors. *SAC 2017*, volume 10719 of *LNCS*. Springer, Heidelberg, August 2017.

[ADMA15] Elena Andreeva, Joan Daemen, Bart Mennink, and Gilles Van Assche. Security of keyed sponge constructions using a modular proof approach. In Gregor Leander, editor, *FSE 2015*, volume 9054 of *LNCS*, pages 364–384. Springer, Heidelberg, March 2015.

[AEL+18] Tomer Ashur, Maria Eichlseder, Martin M. Lauridsen, Gaëtan Leurent, Brice Minaud, Yann Rotella, Yu Sasaki, and Benoît Viguier. Cryptanalysis of MORUS. In Thomas Peyrin and Steven Galbraith, editors, *ASIACRYPT 2018, Part II*, volume 11273 of *LNCS*, pages 35–64. Springer, Heidelberg, December 2018.

[AES01] Advanced Encryption Standard (AES). National Institute of Standards and Technology (NIST), FIPS PUB 197, U.S. Department of Commerce, November 2001.

[AJN16] Jean-Philippe Aumasson, Philipp Jovanovic, and Samuel Neves. Norx v3.0, 2016. Submission to CAESAR, available via https://competitions.cr.yp.to/round3/norxv30.pdf.

[AK18] Ralph Ankele and Stefan Kölbl. Mind the gap - A closer look at the security of block ciphers against differential cryptanalysis. In Carlos Cid and Michael J. Jacobson Jr., editors, *SAC 2018*, volume 11349 of *LNCS*, pages 163–190. Springer, 2018.

[AL18] Ralph Ankele and Eik List. Differential cryptanalysis of round-reduced sparx-64/128. In Bart Preneel and Frederik Vercauteren, editors, *ACNS 18*, volume 10892 of *LNCS*, pages 459–475. Springer, Heidelberg, July 2018.

[ARH+17] Riham AlTawy, Raghvendra Rohit, Morgan He, Kalikinkar Mandal, Gangqiang Yang, and Guang Gong. sLiSCP: Simeck-based permutations for lightweight sponge cryptographic primitives. In Adams and Camenisch [AC17], pages 129–150.

[ARH+18] Riham Altawy, Raghvendra Rohit, Morgan He, Kalikinkar Mandal, Gangqiang Yang, and Guang Gong. SLISCP-light: Towards hardware optimized sponge-specific cryptographic permutations. *ACM Trans. Embed. Comput. Syst.*, 17(4):81:1–81:26, August 2018.

[ATY17] Ahmed Abdelkhalek, Mohamed Tolba, and Amr M. Youssef. Impossible differential attack on reduced round SPARX-64/128. In Marc Joye and Abderrahmane Nitaj, editors, *AFRICACRYPT 17*, volume 10239 of *LNCS*, pages 135–146. Springer, Heidelberg, May 2017.

[BBCdS+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 - (feat. CRAX and TRAX). In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020*, volume 12172 of *LNCS*, pages 419–448. Springer, 2020.

[BBCdS+20b] 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. *IACR Transactions on Symmetric Cryptology*, 2020(S1):208–261, 2020.

[BBD+99] Eli Biham, Alex Biryukov, Orr Dunkelman, Eran Richardson, and Adi Shamir. Initial observations on Skipjack: Cryptanalysis of Skipjack-3XOR (invited talk). In Stafford E. Tavares and Henk Meijer, editors, *SAC 1998*, volume 1556 of *LNCS*, pages 362–376. Springer, Heidelberg, August 1999.

[BBS99] Eli Biham, Alex Biryukov, and Adi Shamir. Cryptanalysis of Skipjack reduced to 31 rounds using impossible differentials. In Jacques Stern, editor, *EUROCRYPT'99*, volume 1592 of *LNCS*, pages 12–23. Springer, Heidelberg, May 1999.

[BDBP03] Alex Biryukov, Christophe De Canniére, An Braeken, and Bart Preneel. A toolbox for cryptanalysis: Linear and affine equivalence algorithms. In Eli Biham, editor, *EUROCRYPT 2003*, volume 2656 of *LNCS*, pages 33–50. Springer, Heidelberg, May 2003.

[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*, volume 9696 of *Lecture Notes in Computer Science*, pages 537–557. Springer, 2016.

[BDP+16a] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. Caesar submission: Ketje v2, 2016. Submission to CAESAR, available via https://competitions.cr.yp.to/round3/ketjev2.pdf.

[BDP+16b] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. Caesar submission: Keyak v2, 2016. Submission to CAESAR, available via https://competitions.cr.yp.to/round3/keyakv22.pdf.

[BDPA11] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Duplexing the sponge: Single-pass authenticated encryption and other applications. In Ali Miri and Serge Vaudenay, editors, *Selected Areas in Cryptography - 18th International Workshop, SAC 2011, Toronto, ON, Canada, August 11-12, 2011, Revised Selected Papers*, volume 7118 of *Lecture Notes in Computer Science*, pages 320–337. Springer, 2011.

[BDPV08] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. On the indifferentiability of the sponge construction. In Nigel P. Smart, editor, *EUROCRYPT 2008*, volume 4965 of *LNCS*, pages 181–197. Springer, Heidelberg, April 2008.

[BDPVA07] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Sponge functions. In *ECRYPT hash workshop*, 2007.

[BDPVA11] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Cryptographic sponge functions, 2011. Available at https://keccak.team/files/CSF-0.1.pdf.

[BEE+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 Stefan Mangard, editor, *Smart Card Research and Advanced Applications — CARDIS 2012*, volume 7771 of *Lecture Notes in Computer Science*, pages 158–172. Springer Verlag, 2013.

[BKL+17a] 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 Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 299–320. Springer, 2017.

[BKL+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 Wieland Fischer and Naofumi Homma, editors, *CHES 2017*, volume 10529 of *LNCS*, pages 299–320. Springer, Heidelberg, September 2017.

[BR14] Andrey Bogdanov and Vincent Rijmen. Linear hulls with correlation zero and linear cryptanalysis of block ciphers. *Designs, codes and cryptography*, 70(3):369–383, 2014.

[BS91] Eli Biham and Adi Shamir. Differential cryptanalysis of DES-like cryptosystems. In Alfred J. Menezes and Scott A. Vanstone, editors, *CRYPTO'90*, volume 537 of *LNCS*, pages 2–21. Springer, Heidelberg, August 1991.

[BVC16] Alex Biryukov, Vesselin Velichkov, and Yann Le Corre. Automatic search for the best trails in ARX: Application to block cipher speck. In Peyrin [Pey16], pages 289–310.

[BW99] Alex Biryukov and David Wagner. Slide attacks. In Knudsen [Knu99], pages 245–259.

[CCF+21] Tingting Cui, Shiyao Chen, Kai Fu, Meiqin Wang, and Keting Jia. New automatic tool for finding impossible differentials and zero-correlation linear approximations. *Sci. China Inf. Sci.*, 64(2), 2021.

[CDG19] Hao Cheng, Daniel Dinu, and Johann Großschädl. Efficient implementation of the SHA-512 hash function for 8-bit AVR microcontrollers. In Jean-Louis Lanet and Cristian Toma, editors, *Innovative Security Solutions for Information Technology and Communications - 11th International Conference, SecITC 2018, Bucharest, Romania, November 8-9, 2018, Revised Selected Papers*, volume 11359 of *Lecture Notes in Computer Science*, pages 273–287. Springer Verlag, 2019.

[CDNY18] Avik Chakraborti, Nilanjan Datta, Mridul Nandi, and Kan Yasuda. Beetle family of lightweight and secure authenticated encryption ciphers. *IACR TCHES*, 2018(2):218–241, 2018. https://tches.iacr.org/index.php/TCHES/article/view/881.

[CGTV15] Jean-Sébastien Coron, Johann Großschädl, Mehdi Tibouchi, and Praveen Kumar Vadnala. Conversion from arithmetic to boolean masking with logarithmic complexity. In Gregor Leander, editor, *Fast Software Encryption - 22nd International Workshop, FSE 2015, Istanbul, Turkey, March 8-11, 2015, Revised Selected Papers*, volume 9054 of *Lecture Notes in Computer Science*, pages 130–149. Springer, 2015.

[CGV14] Jean-Sébastien Coron, Johann Großschädl, and Praveen Kumar Vadnala. Secure conversion between boolean and arithmetic masking of any order. In Lejla Batina and Matthew Robshaw, editors, *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*, volume 8731 of *Lecture Notes in Computer Science*, pages 188–205. Springer, 2014.

[CN17] Jean-Sébastien Coron and Jesper Buus Nielsen, editors. *EUROCRYPT 2017, Part III*, volume 10212 of *LNCS*. Springer, Heidelberg, April / May 2017.

[CT16] Jung Hee Cheon and Tsuyoshi Takagi, editors. *ASIACRYPT 2016, Part I*, volume 10031 of *LNCS*. Springer, Heidelberg, December 2016.

[Dae95] Joan Daemen. *Cipher and hash function design strategies based on linear and differential cryptanalysis*. PhD thesis, Doctoral Dissertation, March 1995, KU Leuven, 1995.

[DEMS16] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1.2, 2016. Submission to CAESAR, available via https://competitions.cr.yp.to/round3/asconv12.pdf.

[DES77] Data encryption standard. National Bureau of Standards, NBS FIPS PUB 46, U.S. Department of Commerce, January 1977.

[DKR97] Joan Daemen, Lars R. Knudsen, and Vincent Rijmen. The block cipher Square. In Eli Biham, editor, *FSE'97*, volume 1267 of *LNCS*, pages 149–165. Springer, Heidelberg, January 1997.

[DPU+16] Daniel Dinu, Léo Perrin, Aleksei Udovenko, Vesselin Velichkov, Johann Großschädl, and Alex Biryukov. Design strategies for ARX with provable bounds: Sparx and LAX. In Cheon and Takagi [CT16], pages 484–513.

[DR02] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, 2002.

[Dwo15] Morris J Dworkin. Sha-3 standard: Permutation-based hash and extendable-output functions. Federal Inf. Process. Stds.(NIST FIPS)-202, 2015.

[FWG+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 Peyrin [Pey16], pages 268–288.

[GO18] LLC Gurobi Optimization. Gurobi optimizer reference manual, 2018.

[Hir16] Shoichi Hirose. Sequential hashing with minimum padding. In *NIST Workshop on Lightweight Cryptography 2016*. National Institute of Standards and Technology (NIST), 2016.

[Hir18] Shoichi Hirose. Sequential hashing with minimum padding. *Cryptography*, 2:11, 2018.

[JLM14] Philipp Jovanovic, Atul Luykx, and Bart Mennink. Beyond $2^{c/2}$ security in sponge-based authenticated encryption modes. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part I*, volume 8873 of *LNCS*, pages 85–104. Springer, Heidelberg, December 2014.

[JLM+18] Philipp Jovanovic, Atul Luykx, Bart Mennink, Yu Sasaki, and Kan Yasuda. Beyond conventional security in sponge-based authenticated encryption modes. *Journal of Cryptology*, Jun 2018.

[Knu95] Lars R. Knudsen. Truncated and higher order differentials. In Bart Preneel, editor, *FSE'94*, volume 1008 of *LNCS*, pages 196–211. Springer, Heidelberg, December 1995.

[Knu98] Lars Knudsen. Deal - a 128-bit block cipher. NIST AES Proposal, 1998.

[Knu99] Lars R. Knudsen, editor. *FSE'99*, volume 1636 of *LNCS*. Springer, Heidelberg, March 1999.

[KS07] Liam Keliher and Jiayuan Sui. Exact maximum expected differential and linear probability for 2-round Advanced Encryption Standard. *IET Information Security*, 1(2):53–57, 2007.

[Küç09] Özgül Küçük. The hash function Hamsi. Submission to the NIST SHA-3 competition; available online at https://securewww.esat.kuleuven.be/cosic/publications/article-1203.pdf., 2009.

[KW02] Lars R. Knudsen and David Wagner. Integral cryptanalysis. In Joan Daemen and Vincent Rijmen, editors, *FSE 2002*, volume 2365 of *LNCS*, pages 112–127. Springer, Heidelberg, February 2002.

[LAAZ11] Gregor Leander, Mohamed Ahmed Abdelraheem, Hoda AlKhzaimi, and Erik Zenner. A cryptanalysis of PRINTcipher: The invariant subspace attack. In Phillip Rogaway, editor, *CRYPTO 2011*, volume 6841 of *LNCS*, pages 206–221. Springer, Heidelberg, August 2011.

[Liu17] Zhengbin Liu. *Automatic Tools for Differential and Linear Cryptanalysis of ARX Ciphers*. PhD thesis, University of Chinese Academy of Science, 2017. In Chinese.

[LLJW21] Zhengbin Liu, Yongqiang Li, Lin Jiao, and Mingsheng Wang. A new method for searching optimal differential and linear trails in arx ciphers. *IEEE Transactions on Information Theory*, 67(2):1054–1068, 2021.

[LMM91] Xuejia Lai, James L. Massey, and Sean Murphy. Markov ciphers and differential cryptanalysis. In Donald W. Davies, editor, *EUROCRYPT'91*, volume 547 of *LNCS*, pages 17–38. Springer, Heidelberg, April 1991.

[LWR16] Yunwen Liu, Qingju Wang, and Vincent Rijmen. Automatic search of linear trails in ARX with applications to SPECK and chaskey. In Mark Manulis, Ahmad-Reza Sadeghi, and Steve Schneider, editors, *ACNS 16*, volume 9696 of *LNCS*, pages 485–499. Springer, Heidelberg, June 2016.

[MA20] Amirhossein Ebrahimi Moghaddam and Zahra Ahmadian. New automatic search method for truncated-differential characteristics application to Midori, SKINNY and CRAFT. *The Computer Journal*, 63(12):1813–1825, 2020.

[Mat94] Mitsuru Matsui. Linear cryptanalysis method for DES cipher. In Tor Helleseth, editor, *EUROCRYPT'93*, volume 765 of *LNCS*, pages 386–397. Springer, Heidelberg, May 1994.

[Mat95] Mitsuru Matsui. On correlation between the order of S-boxes and the strength of DES. In Alfredo De Santis, editor, *EUROCRYPT'94*, volume 950 of *LNCS*, pages 366–375. Springer, Heidelberg, May 1995.

[NPB15] Aina Niemetz, Mathias Preiner, and Armin Biere. Boolector 2.0 system description. *Journal on Satisfiability, Boolean Modeling and Computation*, 9:53–58, 2014 (published 2015).

[Pey16] Thomas Peyrin, editor. *FSE 2016*, volume 9783 of *LNCS*. Springer, Heidelberg, March 2016.

[PGC98] Jacques Patarin, Louis Goubin, and Nicolas Courtois. Improved algorithms for isomorphisms of polynomials. In Kaisa Nyberg, editor, *EUROCRYPT'98*, volume 1403 of *LNCS*, pages 184–200. Springer, Heidelberg, May / June 1998.

[QSLG17] Kexin Qiao, Ling Song, Meicheng Liu, and Jian Guo. New collision attacks on round-reduced keccak. In Coron and Nielsen [CN17], pages 216–243.

[SBK+17] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full SHA-1. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part I*, volume 10401 of *LNCS*, pages 570–596. Springer, Heidelberg, August 2017.

[SLG17] Ling Song, Guohong Liao, and Jian Guo. Non-full sbox linearization: Applications to collision attacks on round-reduced keccak. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part II*, volume 10402 of *LNCS*, pages 428–451. Springer, Heidelberg, August 2017.

[SMG15] Tobias Schneider, Amir Moradi, and Tim Güneysu. Arithmetic addition over boolean masking - towards first- and second-order resistance in hardware. In Tal Malkin, Vladimir Kolesnikov, Allison Bishop Lewko, and Michalis Polychronakis, editors, *ACNS 2015*, volume 9092 of *LNCS*, pages 559–578. Springer, 2015.

[ST17] Yu Sasaki and Yosuke Todo. New impossible differential search tool from design and cryptanalysis aspects - revealing structural properties of several ciphers. In Coron and Nielsen [CN17], pages 185–215.

[SWW17] Ling Sun, Wei Wang, and Meiqin Wang. Automatic search of bit-based division property for ARX ciphers and word-based division property. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 128–157. Springer, Heidelberg, December 2017.

[SWW20] Ling Sun, Wei Wang, and Meiqin Wang. Milp-aided bit-based division property for primitives with non-bit-permutation linear layers. *IET Information Security*, 14(1):12–20, 2020.

[TAY17] Mohamed Tolba, Ahmed Abdelkhalek, and Amr M. Youssef. Multidimensional zero-correlation linear cryptanalysis of reduced round SPARX-128. In Adams and Camenisch [AC17], pages 423–441.

[TLS16] Yosuke Todo, Gregor Leander, and Yu Sasaki. Nonlinear invariant attack - practical attack on full SCREAM, iSCREAM, and Midori64. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016, Part II*, volume 10032 of *LNCS*, pages 3–33. Springer, Heidelberg, December 2016.

[TM16] Yosuke Todo and Masakatu Morii. Bit-based division property and application to simon family. In Peyrin [Pey16], pages 357–377.

[Tod15] Yosuke Todo. Structural evaluation by generalized integral property. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 287–314. Springer, Heidelberg, April 2015.

[VBCG14] S. Varrette, P. Bouvry, H. Cartiaux, and F. Georgatos. Management of an academic hpc cluster: The ul experience. In *Proc. of the 2014 Intl. Conf. on High Performance Computing & Simulation (HPCS 2014)*, pages 959–967, Bologna, Italy, July 2014. IEEE.

[VV17] Serge Vaudenay and Damian Vizár. Under pressure: Security of caesar candidates beyond their guarantees. Cryptology ePrint Archive, Report 2017/1147, 2017. https://eprint.iacr.org/2017/1147.

[Wag99] David Wagner. The boomerang attack. In Knudsen [Knu99], pages 156–170.

[WT86] A. F. Webster and Stafford E. Tavares. On the design of S-boxes (impromptu talk). In Hugh C. Williams, editor, *CRYPTO'85*, volume 218 of *LNCS*, pages 523–534. Springer, Heidelberg, August 1986.

[XZBL16] Zejun Xiang, Wentao Zhang, Zhenzhen Bao, and Dongdai Lin. Applying MILP method to searching integral distinguishers based on division property for 6 lightweight block ciphers. In Cheon and Takagi [CT16], pages 648–678.

[YZS+15] Gangqiang Yang, Bo Zhu, Valentin Suder, Mark D. Aagaard, and Guang Gong. The simeck family of lightweight block ciphers. In Tim Güneysu and Helena Handschuh, editors, *CHES 2015*, volume 9293 of *LNCS*, pages 307–329. Springer, Heidelberg, September 2015.

[ZR19] Wenying Zhang and Vincent Rijmen. Division cryptanalysis of block ciphers with a binary diffusion layer. *IET Information Security*, 13(2):87–95, 2019.

A 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;
  }
}
```

B Linear Trails in Alzette

Table B.1: The input and output masks α, β (in hex) of all linear trails over Alzette corresponding to maximum expected absolute linear trail correlation $c = 2^{-2}$ and $c = 2^{-5}$ for four and five rounds, respectively. The column $\max\{-\log_2(\tilde{c})\}$ represents the smallest observed correlations of the approximations taken over *all* (combinations of) Alzette instances that can occur without a step counter addition. Similarly, the column $\min\{-\log_2(\tilde{c})\}$ represents the largest observed correlations of the approximations. In all of the experiments, the sample size was 2^{24}.

rounds	α	β	$-\log_2(\lvert c\rvert)$	$\max\{-\log_2(\lvert\tilde{c}\rvert)\}$	$\min\{-\log_2(\lvert\tilde{c}\rvert)\}$
4	0000030180020100	c001018101800001	2.00	2.00	2.00
	0000030180020100	800101c101c00001	2.00	2.00	2.00
	0000020180020180	800101c101c00001	2.00	2.00	2.00
	0000020180020180	c001018101800001	2.00	2.00	2.00
5	0000020180020180	01c00181c1808081	5.00	5.62	5.49
	0000030180020100	01c081c1c180c081	5.00	5.60	5.47
	0000020180020180	01c081c1c180c081	5.00	5.59	5.51
	0000030180020100	41c00101c18080c1	5.00	5.60	5.48
	0000020180020180	41c00101c18080c1	5.00	5.60	5.48
	0000020180020180	41c08141c180c0c1	5.00	5.61	5.48
	0000020180020180	01e08141e180c0c1	5.00	5.59	5.49
	0000030180020100	41c08141c180c0c1	5.00	5.61	5.49
	0000030180020100	01e08141e180c0c1	5.00	5.60	5.47
	0000020180020180	01e00101e18080c1	5.00	5.61	5.50
	0000030180020100	41e00181e1808081	5.00	5.61	5.48
	0000020180020180	41e081c1e180c081	5.00	5.61	5.49
	0000030180020100	01e00101e18080c1	5.00	5.61	5.49
	0000020180020180	41e00181e1808081	5.00	5.61	5.48
	0000030180020100	41e081c1e180c081	5.00	5.61	5.50
	0000030180020100	01c00181c1808081	5.00	5.61	5.49

Table B.2: The input and output masks α, β (in hex) of all linear trails over Alzette corresponding to maximum expected absolute linear trail correlation $c = 2^{-8}$ for six rounds. The column $\max\{-\log_2(\tilde{c})\}$ represents the smallest observed correlations of the approximations taken over *all* combinations of Alzette instances that can occur without a step counter addition. Similarly, the column $\min\{-\log_2(\tilde{c})\}$ represents the largest observed correlations of the approximations. In all of the experiments, the sample size was 2^{28}.

rounds	α	β	$-\log_2(\lvert c\rvert)$	$\max\{-\log_2(\lvert\tilde{c}\rvert)\}$	$\min\{-\log_2(\lvert\tilde{c}\rvert)\}$
6	0000020180020180	05638604c3828201	8.00	9.61	8.50
	0000030180020100	05638604c3828201	8.00	9.69	8.48
	0000020180020180	05c38604c3828241	8.00	8.69	8.00
	0000020180020180	04838604c3828281	8.00	9.20	8.22
	0000020180020180	06038604c3828381	8.00	9.09	8.23
	0000030180020100	05c38604c3828241	8.00	8.71	8.01
	0000030180020100	04838604c3828281	8.00	9.08	8.25
	0000030180020100	06038604c3828381	8.00	9.14	8.23
	0000020180020180	05638484c2828201	8.00	9.69	8.48
	0000020180020180	05c38484c2828241	8.00	8.69	8.01
	0000020180020180	04838484c2828281	8.00	9.17	8.26
	0000020180020180	06038484c2828381	8.00	9.10	8.21
	0000020180020180	05c3c404e2828241	8.00	9.65	8.48
	0000030180020100	07438604c3828301	8.00	9.12	8.24
	0000020180020180	07438604c3828301	8.00	9.10	8.20
	0000030180020100	05638484c2828201	8.00	9.59	8.49
	0000030180020100	05c38484c2828241	8.00	8.74	8.03
	0000030180020100	07e38484c2828301	8.00	9.69	8.47
	0000030180020100	07438484c2828341	8.00	8.71	8.01
	0000030180020100	04838484c2828281	8.00	9.08	8.23
	0000030180020100	07438484c2828301	8.00	9.11	8.23
	0000020180020180	07e38604c3828301	8.00	9.56	8.50
	0000030180020100	05c3c404e2828241	8.00	9.74	8.48
	0000020180020180	0563c404e2828201	8.00	8.70	8.02
	0000030180020100	05c38484c2828201	8.00	9.05	8.25
	0000030180020100	05c38604c3828201	8.00	9.12	8.25
	0000030180020100	06038484c2828381	8.00	9.18	8.24
	0000020180020180	05c3c684e3828241	8.00	9.67	8.51
	0000030180020100	0743c404e2828341	8.00	9.63	8.50
	0000030180020100	0563c404e2828201	8.00	8.73	8.02
	0000030180020100	05c3c684e3828241	8.00	9.70	8.52
	0000030180020100	07e38604c3828301	8.00	9.70	8.49
	0000020180020180	07438484c2828341	8.00	8.69	8.03
	0000020180020180	07438484c2828301	8.00	9.12	8.20
	0000020180020180	05c38604c3828201	8.00	9.09	8.25
	0000020180020180	0743c404e2828341	8.00	9.67	8.47
	0000020180020180	07e3c404e2828301	8.00	8.72	8.01
	0000030180020100	0743c684e3828341	8.00	9.54	8.51
	0000030180020100	0563c684e3828201	8.00	8.76	8.01
	0000030180020100	07e3c684e3828301	8.00	8.72	8.03
	0000020180020180	07e38484c2828301	8.00	9.60	8.51
	0000030180020100	07e3c404e2828301	8.00	8.68	8.01
	0000020180020180	0743c684e3828341	8.00	9.61	8.47
	0000020180020180	0563c684e3828201	8.00	8.74	8.02
	0000020180020180	07438604c3828341	8.00	8.74	8.00
	0000020180020180	05c38484c2828201	8.00	9.06	8.20
	0000030180020100	07438604c3828341	8.00	8.65	8.00
	0000020180020180	07e3c684e3828301	8.00	8.75	8.01

C Representations of the Primitives

The following pages, contain diagrams describing all our algorithms. Their purpose is to help cryptanalysts in their task.

- Figure C.1 contains the absorption of Esch256.
- Figure C.2 contains the absorption of Esch384.
- Figure C.3 contains the encryption of Schwaemm256-128.
- Figure C.4 contains the encryption of Schwaemm192-192.
- Figure C.5 contains the encryption of Schwaemm128-128.
- Figure C.6 contains the encryption of Schwaemm256-256.
- Figure C.7 contains the high-level structure of Sparkle with the notation defined in 4.4.1.

Figure C.1: Esch256 (slim). 3+3 branches, 2 branches rate, 7 steps.

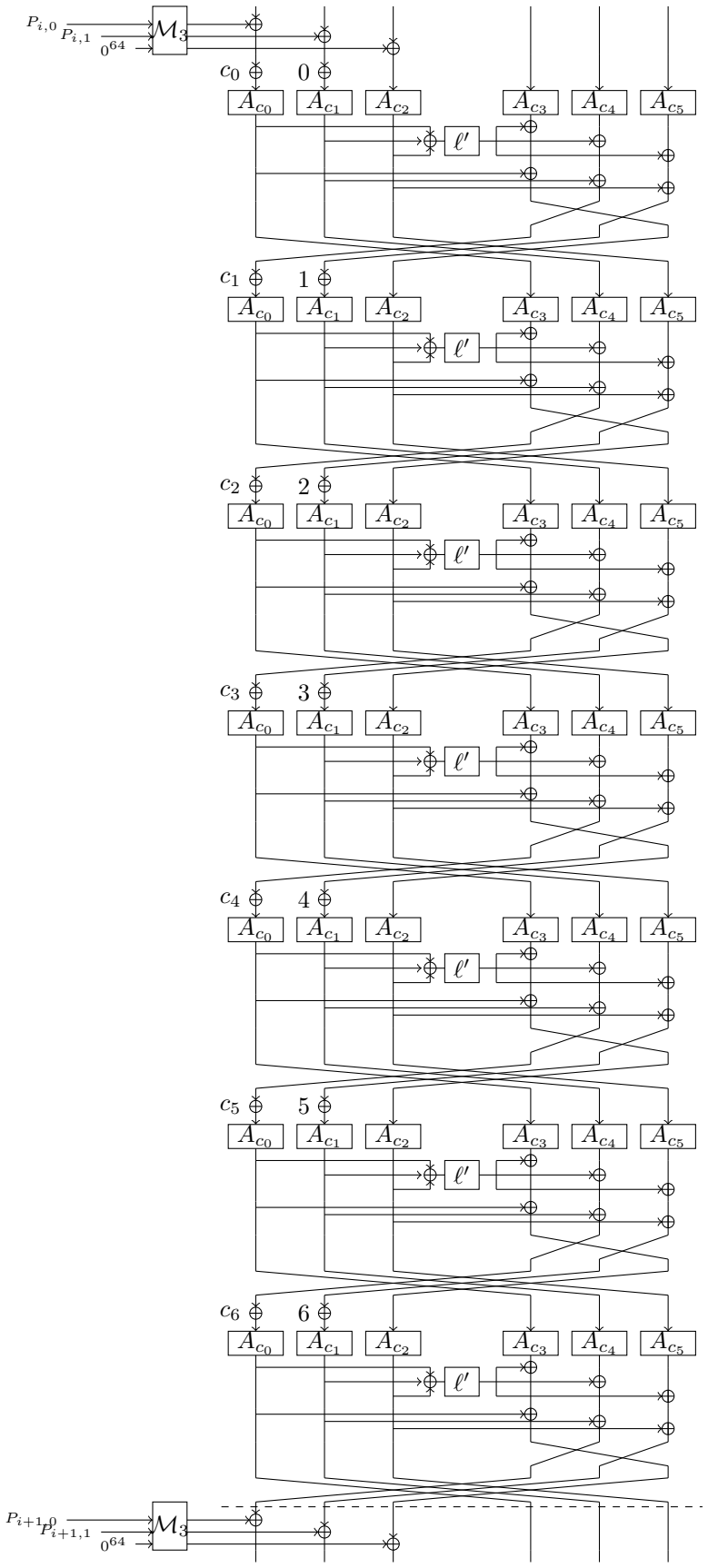


Figure C.2: Esch384 (slim). 4+4 branches, 2 branches rate, 8 steps.

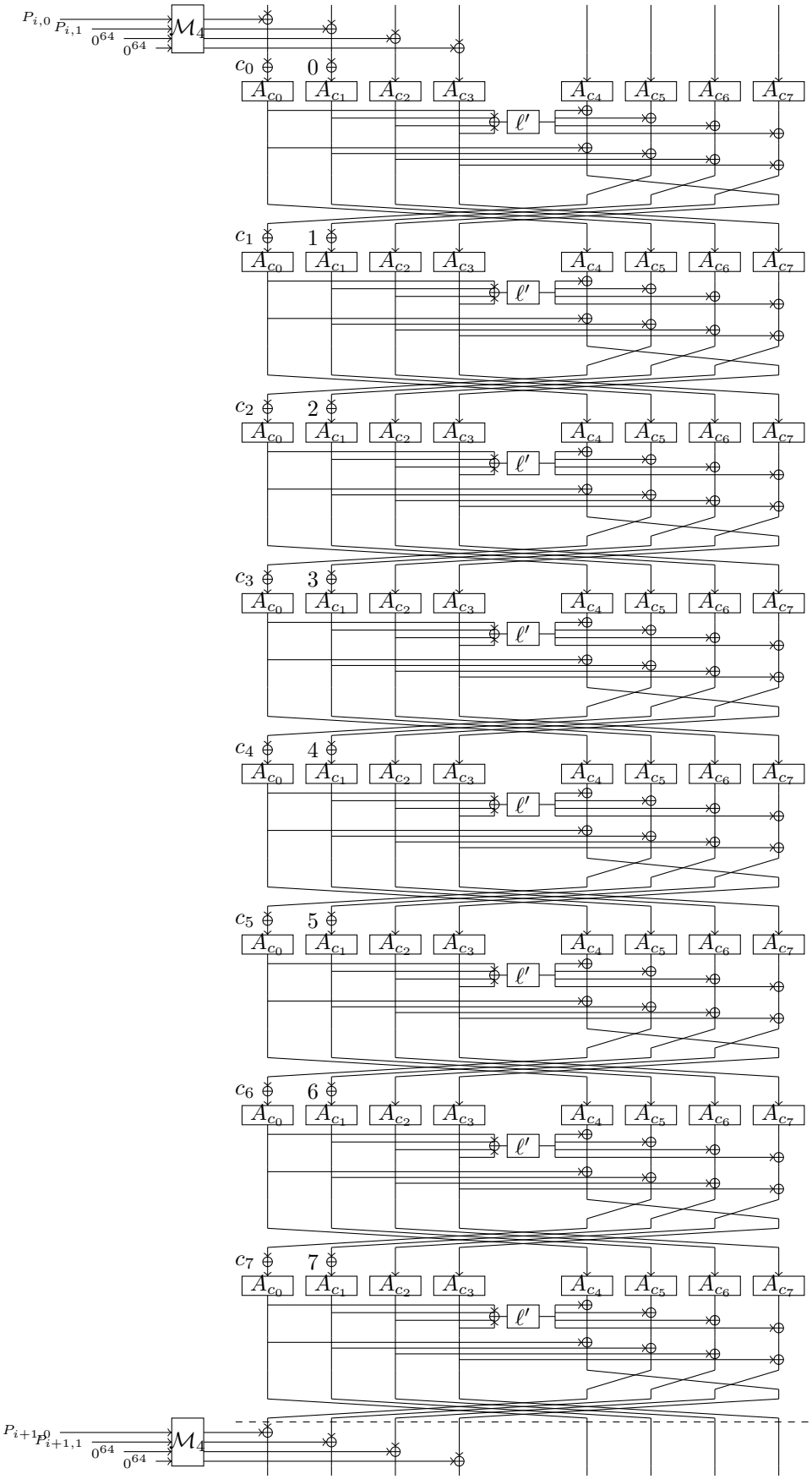


Figure C.3: Schwaemm256-128 (slim). 3+3 branches, 4 branches rate, 7 steps.

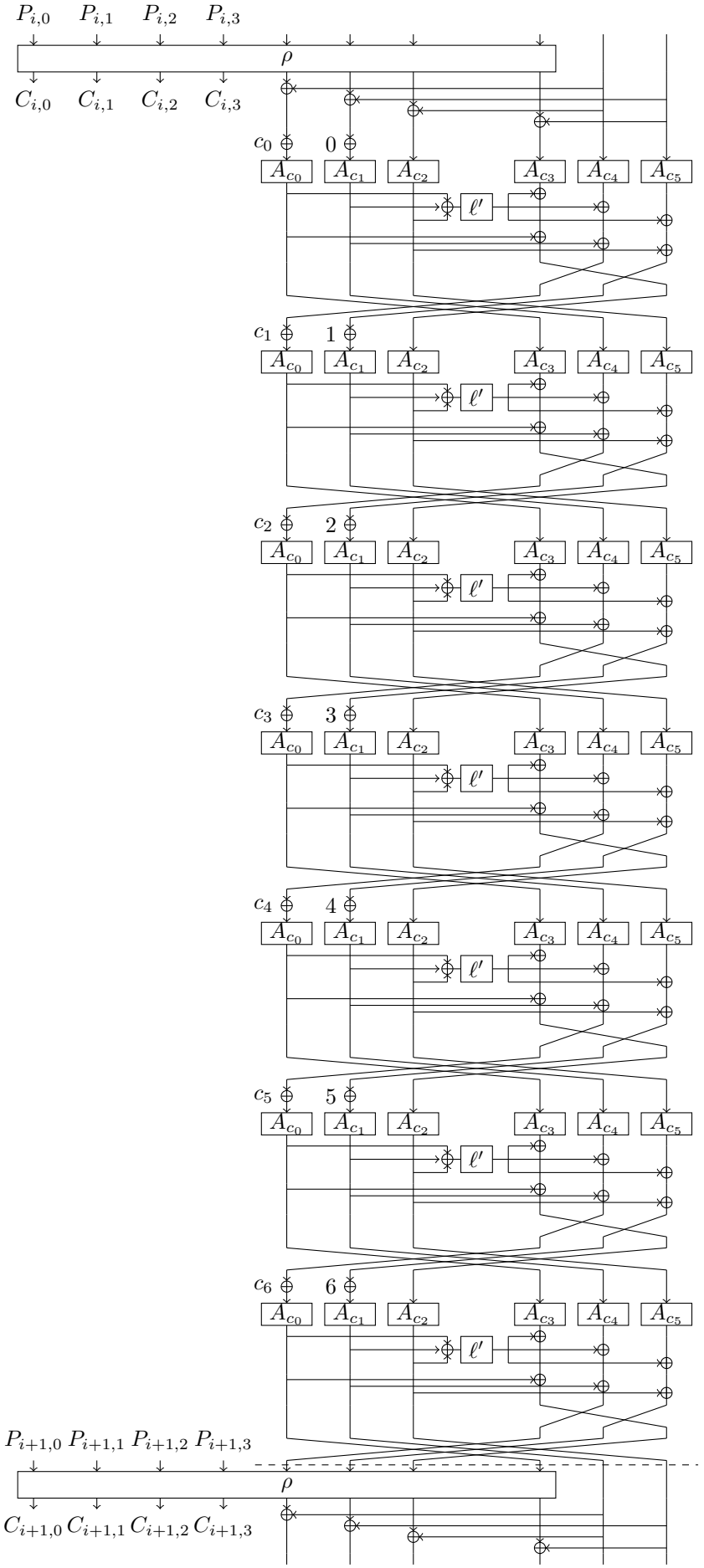


Figure C.4: Schwaemm192-192 (slim). 3+3 branches, 3 branches rate, 7 steps.

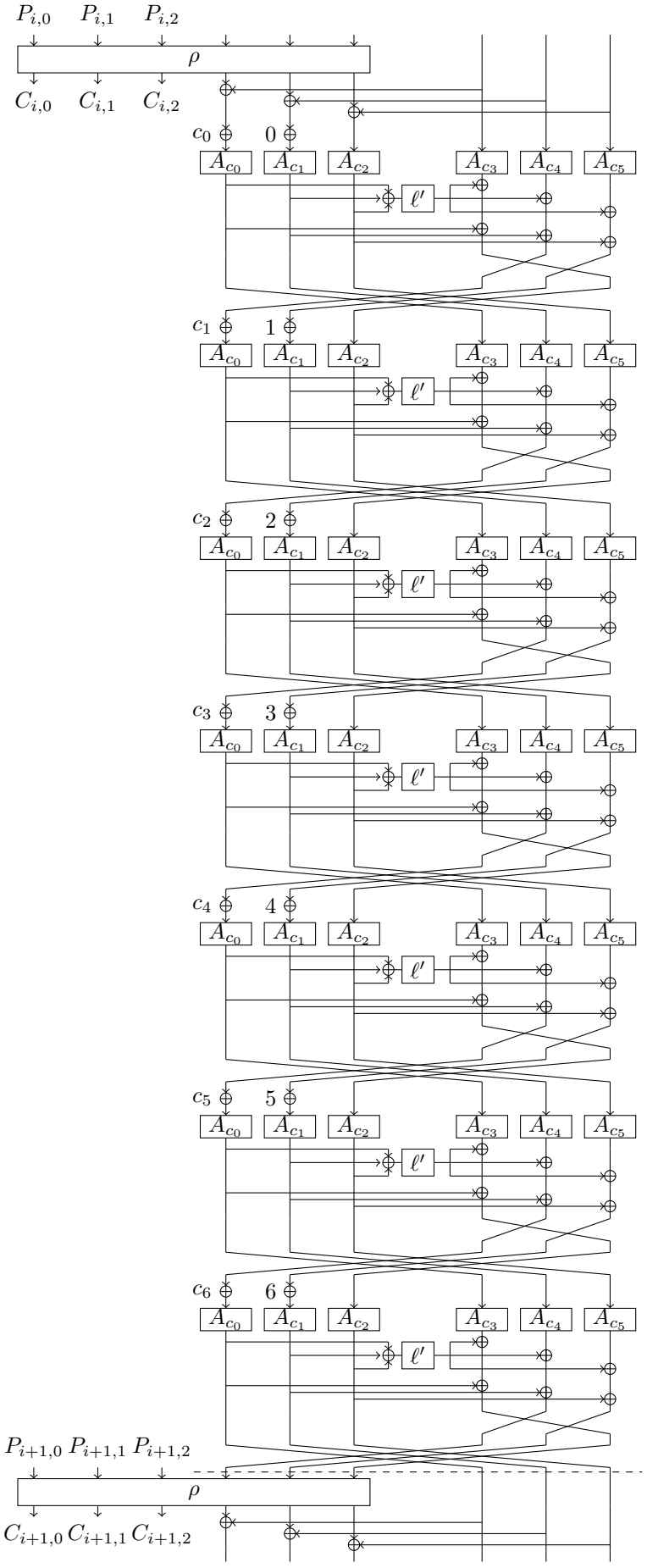


Figure C.5: Schwaemm128-128 (slim). 2+2 branches, 2 branches rate, 7 steps.

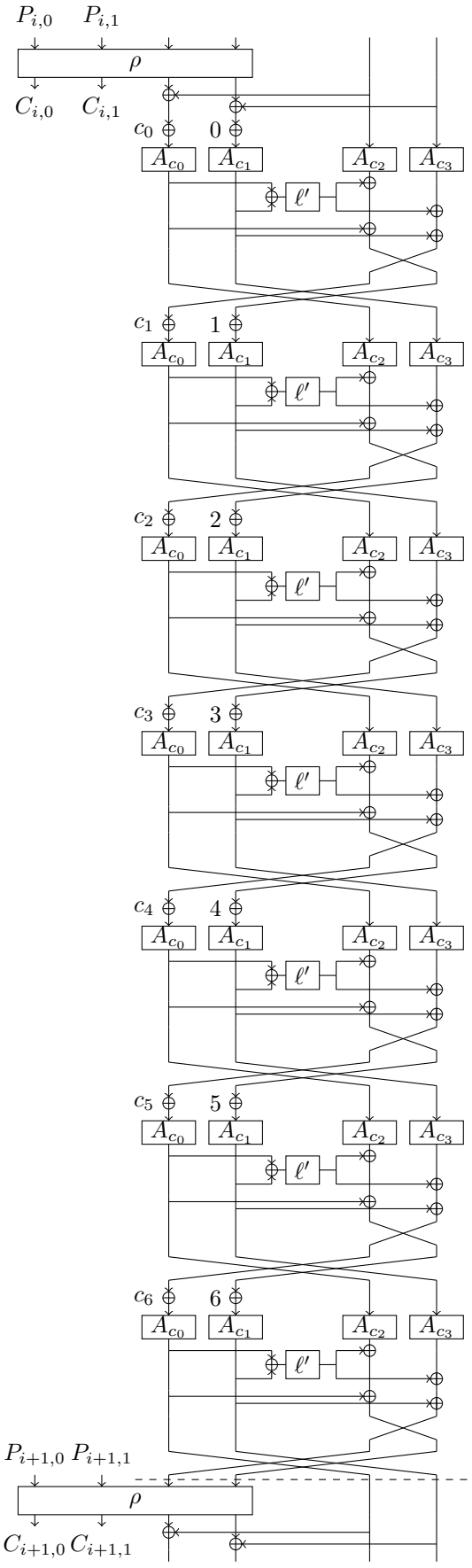


Figure C.6: Schwaemm256-256 (slim). 4+4 branches, 4 branches rate, 8 steps.

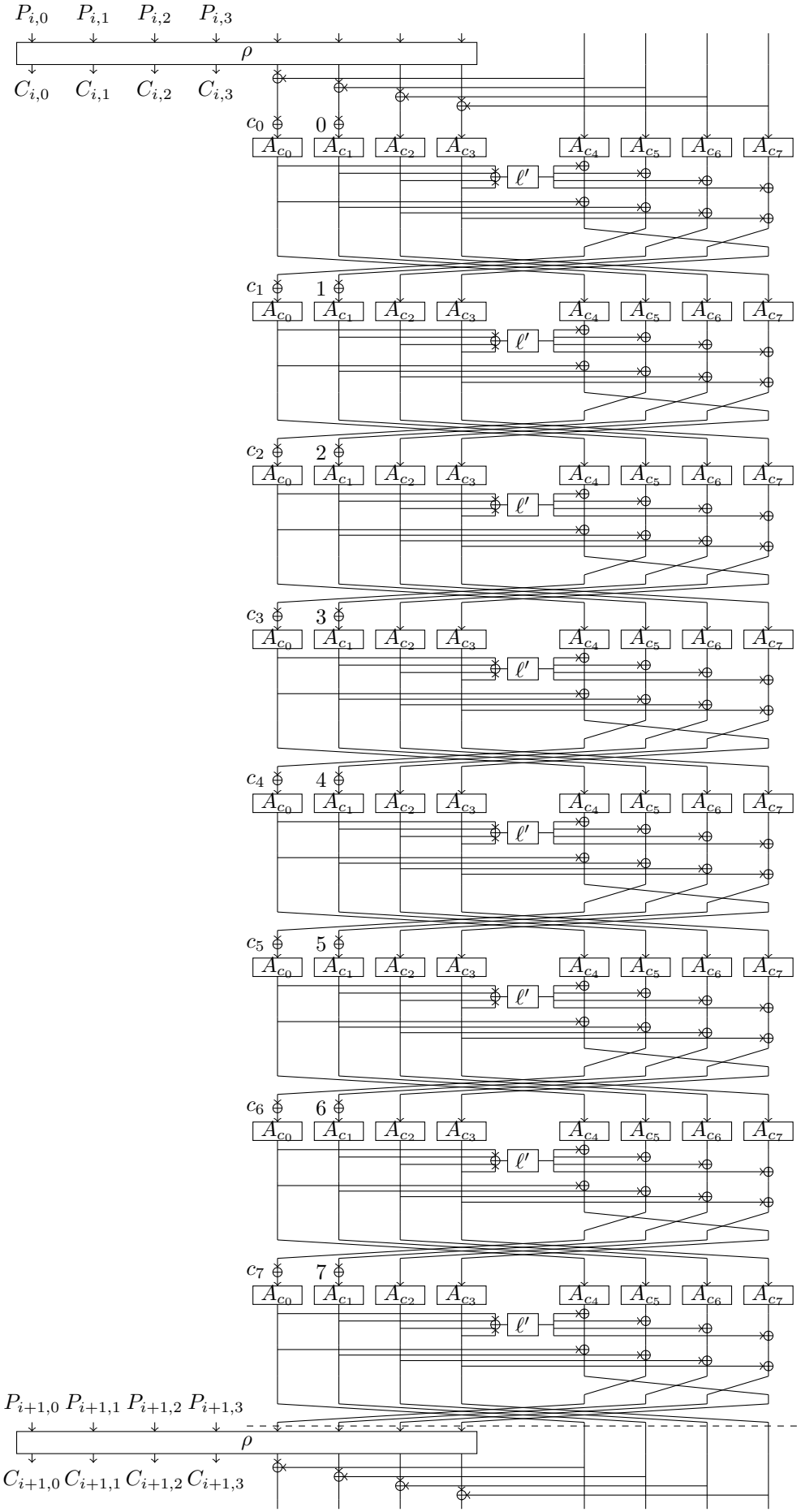


Figure C.7: High-level structure of SPARKLE with 8 steps.