IACR Transactions
ISSN XXXX-XXXX, Vol. 0, No. 0, pp. 1–54. DOI:XXXXXXXX

Lightweight AEAD and Hashing using the Sparkle Permutation Family

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

[1] SnT and CSC, University of Luxembourg, Luxembourg ({first-name.last-name}@uni.lu)
[2] Horst Görtz Institute for IT Security, Ruhr University Bochum, Germany (christof.beierle@rub.de)
[3] Inria, France (leo.perrin@inria.fr)
[4] University of Edinburgh, U.K. (vvelichk@ed.ac.uk)

sparklegrupp@googlegroups.com

Abstract. We introduce the Sparkle family of permutations operating on 256, 384 and 512 bits. These are combined with the Beetle mode to construct a family of authenticated ciphers, Schwaemm, with security levels ranging from 120 to 250 bits. We also use them to build new sponge-based hash functions, Esch256 and Esch384. Our permutations are among those with the lowest footprint in software, without sacrificing throughput. These properties are allowed by our use of an ARX component (the Alzette S-box) as well as a carefully chosen number of rounds. The corresponding analysis is enabled by the long trail strategy which gives us the tools we need to efficiently bound the probability of all the differential and linear trails for an arbitrary number of rounds. We also present a new application of this approach where the only trails considered are those mapping the rate to the outer part of the internal state, such trails being the only relevant trails for instance in a differential collision attack. To further decrease the number of rounds without compromising security, we modify the message injection in the classical sponge construction to break the alignment between the rate and our S-box layer.

Keywords: SPARKLE · NIST · Authenticated Encryption · Hash functions · Lightweight Cryptography · Long Trail Strategy

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 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 National Institute of Standards and Technology (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.

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

Received: 20XX-XX-XX, Accepted: 20XX-XX-XX, Published: 20XX-XX-XX

functions, Esch, are (second) preimage and collision resistant. Our aim for our algorithms is to use as little 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.

1.1 Our Contribution

In this work, we present the ARX-based cryptographic permutation family Sparkle. Using that, we specify the cryptographic hash function family Esch and the authenticated encryption scheme Schwaemm.

Together with the specification of the algorithms (Section 2), we provide a detailed design rationale that explains the choice of the overall structure and its internal components (Section 3). Further, we provide a detailed analysis of the security of our schemes with regard to state-of-the art attacks, in particular differential and linear attacks (Section 4). We further provide details on how the algorithms allow for optimized implementations (Section 5).

One of the main innovations that comes with the design is an application of the *long trail strategy* (LTS) that allows to give bounds on the security against differential and linear attacks *when employing the permutation in a sponge-based construction.* In contrast to many other sponge-based designs, this approach allows for a sound estimation on the number of rounds needed in the permutation. Therefore, our algorithms could be designed with tighter security margins, leading to higher throughputs. We also introduce two simple modifications to the sponge construction which allow us to cheaply increase our security margin: indirect injection (used when hashing) and rate whitening (used in AEAD). Both could be of independent interest.

In the following, we provide some details about the specified algorithms and list their main features with regard to security and efficiency.

Sparkle is closely related to the block cipher Sparx [DPU+16]. 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*.

Esch and Schwaemm are both 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. Esch and Schwaemm can also be well optimized to achieve small silicon area and low power consumption when implemented in hardware. Our schemes are built from well-understood principles, i.e., the sponge (resp. duplex-sponge) construction based on a cryptographic permutation.

We provide two instances of the hash function Esch (i.e., Esch256 which produces a 256-bit digest, offering a security level of 128 bits, and Esch384 which produces a 384-bit digest and offers a security level of 192 bits). These serve as the basis for two Extendable-Output Functions (XOFs): XOEsch256 and XOEsch384.

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 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.

1.2 Key Features

High Efficiency. Both Schwaemm and Esch are characterized by a relatively small state size, which is only 256 bits for the most lightweight instance of Schwaemm 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.

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 [CDG19]. 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 (see [BBdS+19]). 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.

Schwaemm and Esch were designed to be consistent across security levels, which facilitates a parameterized software implementation of the algorithms and the underlying permutation. 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.

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.

High Security. We have not traded security for efficiency. Our detailed security analysis 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. 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.[1]

The design of the Sparkle family of permutations is based on an 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 and a linear permutation with a high (differential and linear) branch number. To combine these two types of subcomponents, we rely on the design strategy that was used for the block cipher Sparx: the LTS. Our substitution layer operates on 64-bit branches using ARX-based S-boxes. The fact that the block size of the ARX component (the *ARX-box*, named Alzette [BBdS+19][2]) 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.

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 recently announced winner of the CAESAR competition Ascon [DEMS16] and the third-round CAESAR candidate Ketje [BDP+16], both authenticated ciphers, use such an approach. The advantage in this case is a much higher efficiency as we need fewer rounds of the permutation. However, the security guarantees are *a priori* weaker in this case as it is harder to estimate the strength needed by the permutation. It is necessary to carefully assess the security of the specific permutation used with the knowledge of the mode of operation it is intended for.

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 LTS, 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.

2 Specification

We make no distinction between the sets $\mathbb{F}_2^{a+b}$ and $\mathbb{F}_2^a \times \mathbb{F}_2^b$ and interpret those to be the same. However, 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 byte order are assumed to be little-endian.

[1] The advantage of the Beetle mode compared to a simple duplexed sponge is that it 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] Alzette is pronounced [alzɛt].

The specification of the SPARKLE permutation and of its various instances is given in Section 2.1. We use these permutations to specify the hash functions ESCH in Section 2.2 and the authenticated ciphers SCHWAEMM in Section 2.4.

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

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 [BBdS+19] (denoted A), i.e., a 64-bit block cipher with a 32-bit key
$$A\colon (\mathbb{F}_2^{32} \times \mathbb{F}_2^{32}) \times \mathbb{F}_2^{32} \to (\mathbb{F}_2^{32} \times \mathbb{F}_2^{32}), ((x, y), c) \mapsto (u, v)\ .$$
We define A_c to be the permutation $(x, y) \mapsto A(x, y, c)$ from $\mathbb{F}_2^{32} \times \mathbb{F}_2^{32}$ to $\mathbb{F}_2^{32} \times \mathbb{F}_2^{32}$.
- A linear *diffusion layer* $\mathcal{L}_{n_b}\colon \mathbb{F}_2^{64n_b} \to \mathbb{F}_2^{64n_b}$, where n_b denotes the number of 64-bit branches, i.e., the block size divided by 64. It is necessary that n_b is even.

The high-level structure of the permutations is given in Algorithms 1 (in this section), 9 and 10 (in Appendix A). 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 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 1. Before each step, a sparse step-dependent constant is XORed to the cipher's state (i.e., to y_0 and y_1).

Algorithm 1 SPARKLE256$_{n_s}$
In/Out: $\big((x_0, y_0), ..., (x_3, y_3)\big), x_i, y_i \in \mathbb{F}_2^{32}$

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

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

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

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. Our motivation for this difference is given in Section 3.5. The slim and big versions of all SPARKLE instances are given in Table 1.

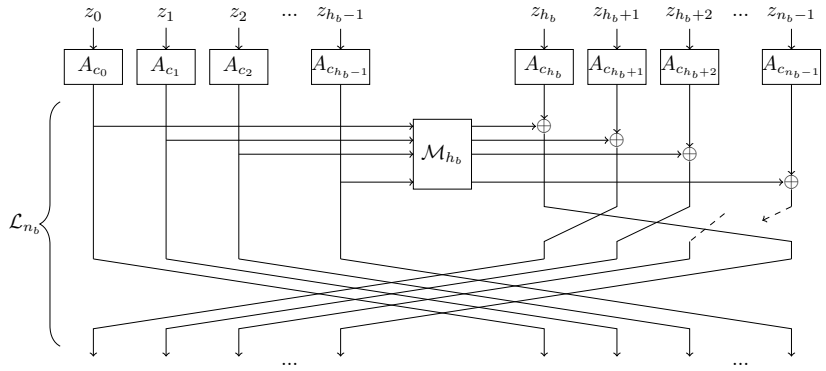

Figure 1: The overall structure of a step of SPARKLE. z_i denotes the 64-bit input (x_i, y_i).

2.1.1 The ARX-box Alzette

Alzette, shortly denoted A, is a 64-bit block cipher which we presented in [BBdS+19]. It is specified in Algorithm 2 and depicted in Figure 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.

Table 1: The versions of each SPARKLE instance.

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

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.

Algorithm 2 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)

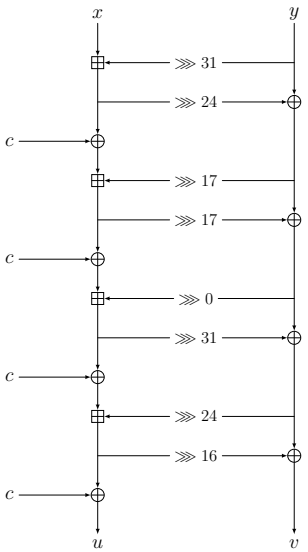

Figure 2: The Alzette instance A_c.

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.

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

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

where each branch (u_i, v_i) is obtained via the following equations

$$u_i \leftarrow x_i \oplus \ell\left(\bigoplus_{i=0}^{w-1} y_i\right) \;, \quad v_i \leftarrow y_i \oplus \ell\left(\bigoplus_{i=0}^{w-1} x_i\right) \;, \tag{1}$$

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 1. Algorithms describing the three diffusion layers used in our permutations are given in Algorithms 11, 12 and 13.

2.2 The Hash Functions Esch256 and Esch384

We propose two instances for hashing, i.e., ESCH256 and ESCH384, which allow to process messages $M \in \mathbb{F}_2^*$ of arbitrary length[3] and output a digest D of bitlengths 256, and 384, respectively. 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 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.

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

The different digest sizes and the corresponding security levels are obtained using different permutation sizes in the sponge, i.e., SPARKLE384_7 and SPARKLE384_{11} for ESCH256 and SPARKLE512_8 and SPARKLE512_{12} for ESCH384. The algorithms are formally specified in Algorithm 4 and 14 and ESCH256 is depicted in Figure 3. 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$_{n_b}$.

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

Table 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.

	n	r	c	collision	2nd preimage	preimage	data limit
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}

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

Algorithm 3 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

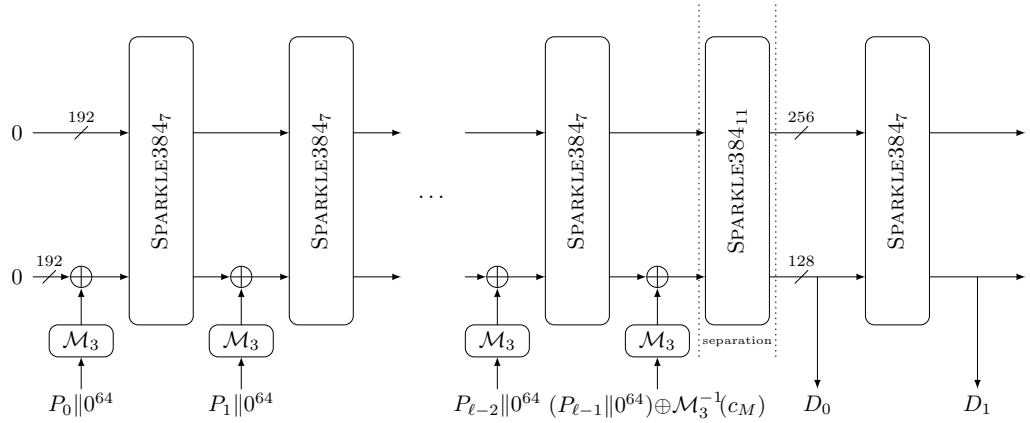

Figure 3: ESCH256 with rate $r = 128$ and capacity $c = 256$. The constant c_M is equal to $(0, 0, \ldots, 0, 1) \in \mathbb{F}_2^{192}$ if the last block was padded and $(0, 0, \ldots, 0, 1, 0) \in \mathbb{F}_2^{192}$ otherwise.

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 5 and 15, respectively. The parameters and security levels are given in Table 2.

Algorithm 4 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 5 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**
  $P'_j \leftarrow \mathcal{M}_3(P_j \| 0^{64})$
  $S \leftarrow \text{SPARKLE384}_7\big(S \oplus (P'_j \| 0^{192})\big)$
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
$D_0 \leftarrow \mathsf{trunc}_{128}(S)$
for all $j = 1, \ldots, \lceil t/128 \rceil - 1$ **do**
  $S \leftarrow \text{SPARKLE384}_7\big(S\big)$
  $D_j \leftarrow \mathsf{trunc}_{128}(S)$
end for
return $\mathsf{trunc}_t(D_0 \| D_1 \| \ldots \| D_{\lceil t/128 \rceil - 1})$

2.4 The Authenticated Cipher Family Schwaemm

We propose four instances for AEAD, 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[4] 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 $\perp$. 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 differences between the instances are the version of the underlying SPARKLE permutation (and thus the rate and the capacity is different) and the size of the 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. 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 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.

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

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

Table 3: The AEAD instances with their (joint) security level in bit with regard to confidentiality and integrity and the limitation in the data to be processed.

	n	r	c	$\|K\|$	$\|N\|$	$\|T\|$	security	data limit (bytes)
SCHWAEMM256-128	384	256	128	128	256	128	120	2^{68}
SCHWAEMM192-192	384	192	192	192	192	192	184	2^{68}
SCHWAEMM128-128	256	128	128	128	128	128	120	2^{68}
SCHWAEMM256-256	512	256	256	256	256	256	248	2^{133}

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 rate, where S_R denotes the internal state corresponding to the inner part. For the SCHWAEMM instances with $r = c$, we define $\mathcal{W}_{c,r}\colon \mathbb{F}_2^c \to \mathbb{F}_2^r$ as the identity (i.e., we just XOR the inner part to the outer part). For SCHWAEMM256-128, we define $\mathcal{W}_{128,256}(x, y) = (x, y, x, y)$, where $x, y \in \mathbb{F}_2^{64}$. Note that this tweak can still be described in the BEETLE framework as the prepended rate whitening can be considered to be part of the definition of the underlying permutation.

Figure 4 depicts the mode for SCHWAEMM256-128. The formal specifications of the encryption and decryption of the four family members are given in Algorithms 6-21.

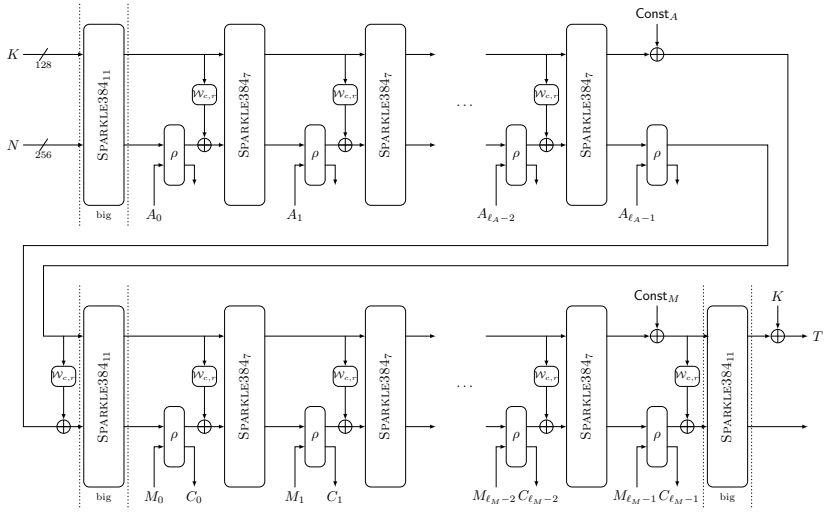

Figure 4: The AEAD Algorithm SCHWAEMM256-128 with $r = 256$ and $c = 128$.

Algorithm 6 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 7 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

3 Design Rationale

In this section, 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[5] (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, generic distinguishers that distinguish the permutation from a random one trivially exist (e.g., the property of being efficiently implementable). When using the term *distinguisher*, 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 Modes of Operation

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 recently 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.

Note that, for the hashing mode as well as for authenticated encryption we use different permutations within each sponge (i.e., slim and big version of SPARKLE). Therefore, the generic provable security argument does not technically apply.

[5] In this paper, we just briefly summarize the properties of Alzette. A detailed design rationale can be found in [BBdS+19].

3.1.2 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.

Rate Whitening In a sponge-based authenticated cipher, the security of the primitive is based on the secrecy of the capacity. Hence, we can safely allow the adversary to read the content of the rate. However, in practice, this can allow the attacker to compute a part of the permutation. Indeed, if the rate 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 capacity remains secret in this case. However, it is easy to prevent this phenomenon using what we call *rate whitening*. It simply consists in XORing branches from the capacity into the rate just before the permutation call. That way, the attacker cannot evaluate a part of the permutation without first guessing parts of the capacity.

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.

Indirect Injection In a sponge-based hash function, an r-bit message block is XORed into the rate. 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 rate 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 rate and the Alzette layer. Furthermore, because the linear functions we use for the indirect injection ($\mathcal{M}_3$ and $\mathcal{M}_4$) have a (differential, resp., linear) branch 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 was established in [BBdS+19], the best differential trail covering a double ARX-box has a probability at most equal to 2^{-32}. We deduce the following lemma.[6]

[6] In the specific cases of the Esch functions, an attacker could try and leverage the padding scheme and

Lemma 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 1 (in Section 3.4.1) shows that no message difference can cancel this constant.

3.2 A Permutation Structure Favouring 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[7] (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) trail/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. squared correlation) of the best differential (resp. linear) trail.

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].

the different constants added in the rate 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.

[7] 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.

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 trail as the product 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 (see [BBdS+19]).

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 trails 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*.

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. First, we recall how a long trail argument works to bound the differential probabilities and absolute linear correlations.

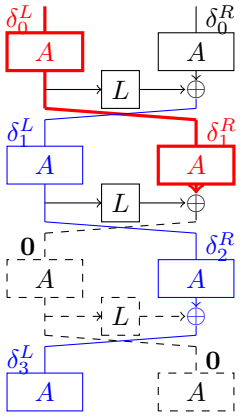

Figure 5: The decomposition into long trails of a truncated trail in a simple cipher.

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 iterating 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 trail 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 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. 5 where the zero difference is dashed). Then this differential trail can be decomposed into 3 long trails represented in black, red and blue: 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.

A good structure to leverage long trail is the one described in Figure 6. 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 layers 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.[8]

These observations led us to use such a linear layer when designing Sparx. Now that Sparx has undergone third-party cryptanalysis [AL18, AK19, ATY17, TAY17] we confidently reuse this structure.

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.

[8] Again, we stress that we assume independent calls to Alzette and use bounds on the MEDCP/MELCC, although we don't have round keys (see experiments in [BBdS+19]).

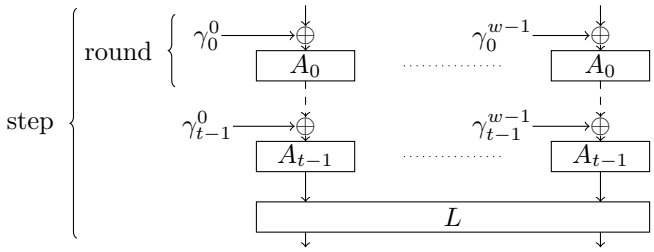

Figure 6: The overall structure of a step for the long trail strategy. The wires correspond to *branches* which, in our case, are divided into two *words*.

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 counterintuitively 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.

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.

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]. 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.

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 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:

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

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 capacity. In other words, Property 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 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

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

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 rate 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 a rate-to-rate 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 exists a valid pair with a fixed and predefined input *value* is $\leq 2^{-\frac{c}{2}}$.

Hypothesis 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 rate-to-rate 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 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 4.2.1.

For Property 2, using an LTS-based permutation simplifies the search greatly. Usually, the search space corresponding to the search for the trails considered in Property 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 2 is a priori impossible to verify unless we simply ensure that Property 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.

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 (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 rate is lower than $2^{-s/2}$, for all numbers of iterations i where i is smaller than the order of the permutation.

Hypothesis 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

A detailed design rationale and security analysis of the ARX-box Alzette is described in [BBdS+19]. Below, we briefly mention its most important properties.

Alzette is constructed from 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.

The rotations were chosen to maximize security and efficiency. While each rotation has the same cost in 32-bit ARM processors, they were chosen to be optimized for implementations on 8 and 16-bit microcontrollers.

Table 4: Bounds for Alzette compared to Speck64 [BSS+13]. The first line shows $-\log_2 p$, where p is the maximum expected differential trail probability and the second line shows $-\log_2 c$, where c is the maximum expected absolute linear trail correlation. The value set in parenthesis corresponds to the maximum absolute correlation of the linear hull taking clustering into account, derived by experimental verification. For Speck64, the differential bounds are taken from [BVC16] and linear bounds are taken from [FWG+16, LWR16].

	1	2	3	4	5	6	7	8	9	10	11	12
Alzette	0	1	2	6	10	18	≥ 24	≥ 32	≥ 36	≥ 42	≥ 46	≥ 52
	0	0	1	2	5	8	13 (11.64)	17 (15.79)	–	–	–	–
Speck64	0	1	3	6	10	15	21	29	≥ 32	–	–	–
	0	0	1	3	6	9	13	17	19	21	24	27

Differential Properties. The bounds on the maximum expected differential trail probabilities (MEDCP) were computed by a version of Algorithm 1 of [BVC16]. Upon termination, it outputs a trail with the MEDCP. For Alzette, such trails were obtained for up to six rounds, where the 6-round bound is 2^{-18}.

Note that for 7 and 8 rounds, there are no tight bounds due to the high complexity of the search. However, the algorithm exhaustively searched the range up to $-\log_2(p) = 24$ and $-\log_2(p) = 32$ for 7 and 8 rounds respectively, which proves that there are no valid differential trails with an expected differential trail probability larger than 2^{-24} and 2^{-32}, respectively. The bounds are provided in Table 4.

Linear Properties. To get bounds on the maximum expected absolute linear trail correlation (MELCC) of Alzette, the MILP approach described in [FWG+16] was used. It

was feasible to get tight bounds even for 8 rounds, where the 8-round bound for Alzette is 2^{-17}. It was possible to collect all linear trails that correspond to the MELCC for 4 up to to 8 rounds and to experimentally check the actual correlations of the corresponding linear approximations. While there was only negligible clustering in the differential case, slight clustering could be observed here (see bounds in Table 4).

Diffusion. All output bits depend on all the input bits after one iteration of Alzette, though this dependency can be very weak. After two iterations, 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.

Round Constants. The purpose of round constant additions is to ensure that the Alzette instances called on each branch are independent to avoid symmetries.

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. For more on that, see [BBdS+19].

3.4 The Linear Layer

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.

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

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

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

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

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

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

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

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

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. We deduce the following lemma.

Lemma 3. *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 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

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

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 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 $\mathcal{M}_3$ are MDS.*

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 *worst-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 rate-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 Sparkle512, respectively, in order to prevent differential and linear distinguishers based on the bounds of the LTS (see Tables 6 and 7). 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 8 corresponding to the "rate-to-anything" trails, i.e., where the input difference is constrained to be in the rate 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 rate-to-rate trails (where the rate is always considered to be $\frac{b}{2}$ because of the indirect injection). With regard to linear attacks, we use the bounds of the rate-to-rate trails given in Table 9 for Esch. For Schwaemm, we have to use the bounds for the permutation (i.e., Table 7) 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 bits, for Sparkle256, we need five steps to prevent differential and linear attacks. For differential attack, 5 steps are sufficient to prevent rate-to-anything differential trails with a probability under 2^{-128} (see Table 8). 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 $1/c^2$, where c 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 7. 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 rate-to-anything trails, and five steps for rate-to-rate 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 rate-to-rate 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 rate-to-anything bounds. Note that five steps are sufficient if the output difference is in the rate 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 a higher 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, experiments have been made 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), see [BBdS+19]. 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 $\min\left(r, \frac{r+c}{2}, c - \log_2(r)\right)$ 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.[9]

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,[10] 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.

Nonce Misuse Setting. The above security claims are void in cases where nonces are reused. 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 rate 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 rate

[9] 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.

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

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 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 the Sparkle Permutations

We make formal claims only for the big versions. 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}.

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 should not be used for hashing, resp., encryption, they just define possible targets for cryptanalysis.

4.2 Attacks Against the Permutation

We evaluated the security of the SPARKLE permutation family against several attack vectors, listed in Table 5, 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 5, we conducted several statistical tests on the SPARKLE permutations. In particular, we tested for diffusion properties, the distribution of low-weight monomials in the algebraic normal form and the resistance against slide

attacks. In this paper, we only focus on differential and linear attacks. For the security evaluation with regard to the other attack vectors, we refer to the official NIST submission. Supporting cryptanalysis code can be found at https://github.com/cryptolu/sparkle.

Table 5: 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 guarantee 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 6 and Table 7.

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

4.2.1 Differential Attacks

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 trail using the probability that were established for Alzette (see Section 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 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 $2^{n_b/2} \in \{4, 8, 16\}$.

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 6.

4.2.2 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

Table 6: 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

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

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

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, 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 4.2.1) 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 7.

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 7: 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

4.3 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 6. 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 rate 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 8.

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 8, 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.

Table 8: 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 rate. The $\emptyset$ symbols means that such trails impossible. $r \to r$ denotes "rate to rate" trails and $r \to n$ denotes trails where the input is in the rate 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

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 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 9 bounds such probabilities.

5 Implementation Aspects

5.1 Software Implementations

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

Table 9: The quantity $-\log_2(p)$ where p is an upper bound on the absolute correlation of a linear trail connecting the rate of the input with the rate of the output of various SPARKLE instances. The $\emptyset$ symbols means that the trails connecting the rate 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

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 the 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.

Algorithm 8 The w-branch permutation 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 $(Z_0, ..., Z_{w-1})$

5.1.2 Linear Layer

It is, of course, possible to implement the branch permutation at the end of the linear layer in a straightforward way via a 1-branch left-rotation of the right half, followed by a swap of the left and right branches. However, it is more efficient to combine both operations and implement them with a single loop as shown in Algorithm 8. The optimized C implementation of Sparkle given in Appendix C follows this approach. Both the rotation and swap can be carried out implicitly (and do not cost any clock cycles) when Sparkle is fully unrolled, though this comes at the expense of significantly increased code size. For example, the linear layer of a fully-unrolled implementation of Sparkle384 has an execution time of only 18 clock cycles on a 32-bit ARM Cortex-M3 microcontroller.

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 have a number of properties and features that facilitate efficient hardware implementation, especially when small silicon area is the main design goal. As already mentioned in Section 1.2, an important characteristic of Esch and Schwaemm is the relatively small size of their state (e.g., 256 bits in the case of Schwaemm128-128). A minimalist hardware architecture for Esch or Schwaemm (or both) optimized for small silicon area consists of three main components: (i) a small RAM module (with two 32-bit read ports and a 32-bit write port) that is word-addressable and large enough to hold the state words, (ii) a 32-bit ALU capable to execute all arithmetic/logical operations performed by the Sparkle permutation, and (iii) a control unit, which can, for example, take the form of a hard-wired or a microcode-programmable state machine. Of course, implementations that support both Esch and Schwaemm can share components like the state-RAM or the ALU.

The 32-bit ALU has to be able to execute the following set of basic arithmetic/logical operations: 32-bit XOR, addition of 32-bit words, and rotations of a 32-bit word by four different amounts, namely 16, 17, 24, and 31 bits. Since there are only four different rotation amounts, the rotations can be simply implemented by a collection of 32 4-to-1 multiplexers. There exist a number of different design approaches for a 32-bit adder; 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) or Carry-Skip Adder (CSA), both of which have a delay that grows logarithmically with the word size. On the other hand, when reaching small silicon area is the main goal, one can "re-use" the adder for performing XOR operations. Namely, an RCA can output the XOR of its two inputs by simply suppressing the propagation of carries, which requires an ensemble of 32 AND gates. In summary, a minimalist ALU consists of 32 FA cells, 32 AND gates (to suppress the carries if needed), and 32 4-to-1 multiplexers (for the rotations). To minimize execution time, it makes sense to combine the addition (resp. XOR) with

a rotation into a single operation that can be executed in a single clock cycle. In some sense, the 32-bit ALU for Sparkle can be seen as a stripped-down variant of the ALU of a 32-bit ARM processor whose functionality has been reduced to the six basic operations mentioned at the beginning of this paragraph.

5.3 Protection against Side-Channel Attacks

A straightforward implementation of a symmetric cryptosystem such as Schwaemm is normally vulnerable to side-channel attacks, in particular to Differential Power Analysis (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, 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 in the end to yield the correct result.

Depending on the actual operation to be protected against DPA, a masking scheme can be Boolean (using logical XOR), arithmetic (using modular addition or modular subtraction) or multiplicative (using modular multiplication). Since Schwaemm is an ARX design and, consequently, involves arithmetic and Boolean operations, the masks have to be converted from one form to the other without introducing any kind of leakage. There exists an abundant literature on mask conversion techniques and it is nowadays well understood how one can convert efficiently from arithmetic masks to Boolean masks and vice versa, see e.g. [CGV14]. An alternative approach is to compute the arithmetic operations (i.e. modular addition) directly on Boolean shares as described in e.g. [CGTV15]. The development of masked implementations of Schwaemm in hardware and software, as well as detailed analysis of the performance impact of masking, is part of our future work.

5.4 Implementation Results

We developed reference and optimized C implementations of all 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 with respect to 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 one and the same assembler function is capable to support both the slim and big number of steps specified in Table 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 simply 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 10. 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.

Table 10: Performance of the Sparkle permutations 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 (including function-call overhead).

Permutation	Rounds	AVR		ARM	
		C	asm	C	asm
Sparkle256	7 (slim)	697 (22305)	179 (5728)	46 (1487)	19 (615)
	10 (big)	992 (31761)	254 (8146)	66 (2111)	27 (858)
Sparkle384	7 (slim)	680 (32679)	173 (8318)	45 (2173)	20 (935)
	11 (big)	1066 (51215)	271 (13022)	71 (3397)	30 (1435)
Sparkle512	8 (slim)	768 (49169)	194 (12454)	51 (3263)	24 (1529)
	12 (big)	1150 (73633)	291 (18638)	76 (4879)	35 (2269)

The performance gap between the compiled C code and the hand-written assembler code is a bit smaller on ARM, namely 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 48 bytes on the stack, of which 40 bytes are due to callee-saved registers (see Table 11). All execution times for ARM specified in Table 10 were obtained with the cycle-accurate instruction set simulator of Keil MicroVision using a generic Cortex-M3 model as target device[11]. It should be noted that these ARM assembler implementations are optimized to achieve a balance between small

[11] 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 10 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+17].

code size and high speed, which means we refrained from optimization techniques that would increase the code size significantly, like full loop unrolling (i.e. unrolling not only the branches but also the steps). An aggressively speed-optimized assembler implementation with fully unrolled loops can be a bit faster, not only due to the elimination of the overhead of the step-loop, but also because the execution time of the linear layer can be further reduced (concretely, the 1-branch left-rotation of the right-side branches in the linear layer could be done "implicitly").

Table 11: 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	48

Table 12: 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)	1928	494	30.88
GIMLI (24 rounds)	3950	1041	65.06
SPARKLE384 (7 steps)	2820	781	24.40
XOODOO (12 rounds)	2376	657	27.38

Besides SPARKLE a multitude of other permutation-based designs were 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 12 compares the execution time and code size of the permutations of ASCON, GIMLI and XOODOO with a fully-unrolled version of SPARKLE384, the main instance of SPARKLE[12]. The full loop unrolling reduces the execution time of SPARKLE384 from 935 to 781 clock cycles, but this reduction by 154 cycles comes at the expense of an almost six-fold increase of code size. Also given in Table 12 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 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 13 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 14 summarizes the corresponding results for an ARM Cortex-M3 device.

[12] 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 13: 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 14: 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)

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. 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 15 summarizes the implementation results of Esch256, SHA-2, SHA-3, some SHA-3 finalists, as well as Gimli[BKL+17]. 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 assembler code only for the Sparkle permutation. We expect that a fully-optimized implementation of Esch256 with all its components written in assembler has the potential to be faster than Blake-256 and get very close to (or even outperform) SHA-256. Such a "pure" assembler implementation of Esch256 is part of our future work and will be made available on the Sparkle homepage at http://cryptolux.org/index.php/Sparkle.

A comparison of the performance of hash functions is easily possible because there exists a number of implementation results in the literature (e.g. [BEE+13]) that were obtained in a consistent fashion. In particular, 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. Unfortunately, there seems to be no

Table 15: 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+17]	1610	778*
Gimli-Hash fast	[BKL+17]	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$[r = 1088, c = 512]$.

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.

Acknowledgements

Part of 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. Part of the work of Aleksei Udovenko was performed while he was at the University of Luxembourg and 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.

References

[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.

[AK19] 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, Heidelberg, August 2019.

[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.

[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.

[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.

[BBdS+19] Christof Beierle, Alex Biryukov, Luan Cardoso dos Santos, Johann Großschädl, Leo Perrin, Aleksei Udovenko, Vesselin Velichkov, and Qingju Wang. Alzette: A 64-bit ARX-box. Cryptology ePrint Archive, Report 2019/1378, 2019. http://eprint.iacr.org/2019/1378.

[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.

[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+16] 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.

[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.

[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+17] 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.

[BSS+13] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. The SIMON and SPECK families of lightweight block ciphers. Cryptology ePrint Archive, Report 2013/404, 2013. http://eprint.iacr.org/2013/404.

[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.

[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 *LNCS*, 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, *FSE 2015*, volume 9054 of *LNCS*, pages 130–149. Springer, Heidelberg, March 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, *CHES 2014*, volume 8731 of *LNCS*, pages 188–205. Springer, Heidelberg, September 2014.

[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.

[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 Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016, Part I*, volume 10031 of *LNCS*, pages 484–513. Springer, Heidelberg, December 2016.

[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.

[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.

[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.

[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.

[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.

[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.

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

[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.

[TAY17] Mohamed Tolba, Ahmed Abdelkhalek, and Amr M. Youssef. Multidimensional zero-correlation linear cryptanalysis of reduced round SPARX-128. In Carlisle Adams and Jan Camenisch, editors, *SAC 2017*, volume 10719 of *LNCS*, pages 423–441. Springer, Heidelberg, August 2017.

[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.

[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 Lars R. Knudsen, editor, *FSE'99*, volume 1636 of *LNCS*, pages 156–170. Springer, Heidelberg, March 1999.

A Algorithms

- The SPARKLE384 permutation is described in Algorithm 9.
- The SPARKLE512 permutation is described in Algorithm 10.
- The linear layer $\mathcal{L}_4$ is described in Algorithm 11.
- The linear layer $\mathcal{L}_6$ is described in Algorithm 12.
- The linear layer $\mathcal{L}_8$ is described in Algorithm 13.
- The hash function ESCH384 is described in Algorithm 14.
- SCHWAEMM192-192 is described in Algorithm 16 (encryption) and 17 (decryption).
- SCHWAEMM128-128 is described in Algorithm 18 (encryption) and 19 (decryption).
- SCHWAEMM256-256 is described in Algorithm 20 (encryption) and 21 (decryption).

Algorithm 9 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 10 SPARKLE512$_{n_s}$
In/Out: $\big((x_0, y_0), ..., (x_7, y_7)\big), x_i, y_i \in \mathbb{F}_2^{32}$

$(c_0, c_1) \leftarrow$ (`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 $i \in [0, 7]$ **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 $\big((x_0, y_0), ..., (x_7, y_7)\big)$

Algorithm 11 $\mathcal{L}_4$
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 12 $\mathcal{L}_6$
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 13 $\mathcal{L}_8$
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 $\big((x_0, y_0), \ldots, (x_7, y_7)\big)$

Algorithm 14 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 $D_0 \| D_1 \| D_2$

Algorithm 15 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**
  $P'_j \leftarrow \mathcal{M}_4(P_j \| 0^{128})$
  $S \leftarrow \text{SPARKLE512}_8\big(S \oplus (P'_j \| 0^{256})\big)$
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
$D_0 \leftarrow \mathsf{trunc}_{128}(S)$
for all $j = 1, \ldots, \lceil t/128 \rceil - 1$ **do**
  $S \leftarrow \text{SPARKLE512}_8\big(S\big)$
  $D_j \leftarrow \mathsf{trunc}_{128}(S)$
end for
return $\mathsf{trunc}_t(D_0 \| D_1 \| \ldots \| D_{\lceil t/128 \rceil - 1})$

Algorithm 16 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 17 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**
  $A_0 \| A_1 \| \ldots \| A_{\ell_A - 1} \leftarrow A$ 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 $C \neq \epsilon$ **then**
  $C_0 \| C_1 \| \ldots \| C_{\ell_M - 1} \leftarrow C$ with $\forall i \in \{0, \ldots, \ell_M - 2\} : |C_i| = 192$ 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
$S_L \| S_R \leftarrow \text{SPARKLE384}_{11}\big(N \| K\big)$ 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**
    $S_L \| S_R \leftarrow \text{SPARKLE384}_7\big((\rho_1(S_L, A_j) \oplus S_R) \| S_R\big)$
  end for
  ▷ Finalization if ciphertext is empty
  $S_L \| S_R \leftarrow \text{SPARKLE384}_{11}\big((\rho_1(S_L, A_{\ell_A - 1}) \oplus S_R \oplus \mathsf{Const}_A) \| (S_R \oplus \mathsf{Const}_A)\big)$
end if

▷ Decrypting
if $C \neq \epsilon$ **then**
  for all $j = 0, \ldots, \ell_M - 2$ **do**
    $M_j \leftarrow \rho_2'(S_L, C_j)$
    $S_L \| S_R \leftarrow \text{SPARKLE384}_7\big((\rho_1'(S_L, C_j) \oplus S_R) \| S_R\big)$
  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**
    $S_L \| S_R \leftarrow \text{SPARKLE384}_{11}\big((\rho_1(S_L, \mathsf{pad}_{192}(M_{\ell_M - 1})) \oplus S_R \oplus \mathsf{Const}_M) \| (S_R \oplus \mathsf{Const}_M)\big)$
  else
    $S_L \| S_R \leftarrow \text{SPARKLE384}_{11}\big((\rho_1'(S_L, C_{\ell_M - 1}) \oplus S_R \oplus \mathsf{Const}_M) \| (S_R \oplus \mathsf{Const}_M)\big)$
  end if
end if
if $S_R \oplus K = T$ **then**
  return $(M_0 \| M_1 \| \ldots \| M_{\ell_M - 1})$
else
  return $\perp$
end if

Algorithm 18 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
$S_L \| S_R \leftarrow \text{SPARKLE256}_{10}\big(N \| K\big)$ 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**
    $S_L \| S_R \leftarrow \text{SPARKLE256}_7\big((\rho_1(S_L, A_j) \oplus S_R) \| S_R\big)$
  end for
  ▷ Finalization if message is empty
  $S_L \| S_R \leftarrow \text{SPARKLE256}_{10}\big((\rho_1(S_L, A_{\ell_A - 1}) \oplus S_R \oplus \mathsf{Const}_A) \| (S_R \oplus \mathsf{Const}_A)\big)$
end if

▷ Encrypting
if $M \neq \epsilon$ **then**
  for all $j = 0, \ldots, \ell_M - 2$ **do**
    $C_j \leftarrow \rho_2(S_L, M_j)$
    $S_L \| S_R \leftarrow \text{SPARKLE256}_7\big((\rho_1(S_L, M_j) \oplus S_R) \| S_R\big)$
  end for
  $C_{\ell_M - 1} \leftarrow \mathsf{trunc}_t\big(\rho_2(S_L, M_{\ell_M - 1})\big)$
  ▷ Finalization
  $S_L \| S_R \leftarrow \text{SPARKLE256}_{10}\big((\rho_1(S_L, M_{\ell_M - 1}) \oplus S_R \oplus \mathsf{Const}_M) \| (S_R \oplus \mathsf{Const}_M)\big)$
end if

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

Algorithm 19 SCHWAEMM128-128-DEC
Input: (K, N, A, C, T) where $K \in \mathbb{F}_2^{128}$ is a key, $N \in \mathbb{F}_2^{128}$ is a nonce, $A, C \in \mathbb{F}_2^*$ and $T \in \mathbb{F}_2^{128}$
Output: Decryption M of C if the tag T is valid, $\perp$ otherwise

if $A \neq \epsilon$ **then**
  $A_0 \| A_1 \| \ldots \| A_{\ell_A - 1} \leftarrow A$ with $\forall i \in \{0, \ldots, \ell_A - 2\} : |A_i| = 128$ and $1 \leq |A_{\ell_A - 1}| \leq 128$
  if $|A_{\ell_A - 1}| < 128$ **then**
    $A_{\ell_A - 1} \leftarrow \mathsf{pad}_{128}(A_{\ell_A - 1})$
    $\mathsf{Const}_A \leftarrow 0 \oplus (1 \ll 2)$
  else
    $\mathsf{Const}_A \leftarrow 1 \oplus (1 \ll 2)$
  end if
end if
if $C \neq \epsilon$ **then**
  $C_0 \| C_1 \| \ldots \| C_{\ell_M - 1} \leftarrow C$ with $\forall i \in \{0, \ldots, \ell_M - 2\} : |C_i| = 128$ and $1 \leq |C_{\ell_M - 1}| \leq 128$
  $t \leftarrow |C_{\ell_M - 1}|$
  if $|C_{\ell_M - 1}| < 128$ **then**
    $C_{\ell_M - 1} \leftarrow \mathsf{pad}_{128}(C_{\ell_M - 1})$
    $\mathsf{Const}_M \leftarrow 2 \oplus (1 \ll 2)$
  else
    $\mathsf{Const}_M \leftarrow 3 \oplus (1 \ll 2)$
  end if
end if

▷ State initialization
$S_L \| S_R \leftarrow \text{SPARKLE256}_{10}\big(N \| K\big)$ 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**
    $S_L \| S_R \leftarrow \text{SPARKLE256}_7\big((\rho_1(S_L, A_j) \oplus S_R) \| S_R\big)$
  end for
  ▷ Finalization if ciphertext is empty
  $S_L \| S_R \leftarrow \text{SPARKLE256}_{10}\big((\rho_1(S_L, A_{\ell_A - 1}) \oplus S_R \oplus \mathsf{Const}_A) \| (S_R \oplus \mathsf{Const}_A)\big)$
end if

▷ Decrypting
if $C \neq \epsilon$ **then**
  for all $j = 0, \ldots, \ell_M - 2$ **do**
    $M_j \leftarrow \rho_2'(S_L, C_j)$
    $S_L \| S_R \leftarrow \text{SPARKLE256}_7\big((\rho_1'(S_L, C_j) \oplus S_R) \| S_R\big)$
  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**
    $S_L \| S_R \leftarrow \text{SPARKLE256}_{10}\big((\rho_1(S_L, \mathsf{pad}_{128}(M_{\ell_M - 1})) \oplus S_R \oplus \mathsf{Const}_M) \| (S_R \oplus \mathsf{Const}_M)\big)$
  else
    $S_L \| S_R \leftarrow \text{SPARKLE256}_{10}\big((\rho_1'(S_L, C_{\ell_M - 1}) \oplus S_R \oplus \mathsf{Const}_M) \| (S_R \oplus \mathsf{Const}_M)\big)$
  end if
end if
if $S_R \oplus K = T$ **then**
  return $(M_0 \| M_1 \| \ldots \| M_{\ell_M - 1})$
else
  return $\perp$
end if

Algorithm 20 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
$S_L \| S_R \leftarrow \text{SPARKLE512}_{12}\big(N \| K\big)$ 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**
    $S_L \| S_R \leftarrow \text{SPARKLE512}_8\big((\rho_1(S_L, A_j) \oplus S_R) \| S_R\big)$
  end for
  ▷ Finalization if message is empty
  $S_L \| S_R \leftarrow \text{SPARKLE512}_{12}\big((\rho_1(S_L, A_{\ell_A - 1}) \oplus S_R \oplus \mathsf{Const}_A) \| (S_R \oplus \mathsf{Const}_A)\big)$
end if

▷ Encrypting
if $M \neq \epsilon$ **then**
  for all $j = 0, \ldots, \ell_M - 2$ **do**
    $C_j \leftarrow \rho_2(S_L, M_j)$
    $S_L \| S_R \leftarrow \text{SPARKLE512}_8\big((\rho_1(S_L, M_j) \oplus S_R) \| S_R\big)$
  end for
  $C_{\ell_M - 1} \leftarrow \mathsf{trunc}_t\big(\rho_2(S_L, M_{\ell_M - 1})\big)$
  ▷ Finalization
  $S_L \| S_R \leftarrow \text{SPARKLE512}_{12}\big((\rho_1(S_L, M_{\ell_M - 1}) \oplus S_R \oplus \mathsf{Const}_M) \| (S_R \oplus \mathsf{Const}_M)\big)$
end if

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

Algorithm 21 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**
  $A_0 \| A_1 \| \ldots \| A_{\ell_A - 1} \leftarrow A$ 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 $C \neq \epsilon$ **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**
    $C_{\ell_M - 1} \leftarrow \mathsf{pad}_{256}(C_{\ell_M - 1})$
    $\mathsf{Const}_M \leftarrow 2 \oplus (1 \ll 4)$
  else
    $\mathsf{Const}_M \leftarrow 3 \oplus (1 \ll 4)$
  end if
end if

▷ State initialization
$S_L \| S_R \leftarrow \text{SPARKLE512}_{12}\big(N \| K\big)$ 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**
    $S_L \| S_R \leftarrow \text{SPARKLE512}_8\big((\rho_1(S_L, A_j) \oplus S_R) \| S_R\big)$
  end for
  ▷ Finalization if ciphertext is empty
  $S_L \| S_R \leftarrow \text{SPARKLE512}_{12}\big((\rho_1(S_L, A_{\ell_A - 1}) \oplus S_R \oplus \mathsf{Const}_A) \| (S_R \oplus \mathsf{Const}_A)\big)$
end if

▷ Decrypting
if $C \neq \epsilon$ **then**
  for all $j = 0, \ldots, \ell_M - 2$ **do**
    $M_j \leftarrow \rho_2'(S_L, C_j)$
    $S_L \| S_R \leftarrow \text{SPARKLE512}_8\big((\rho_1'(S_L, C_j) \oplus S_R) \| S_R\big)$
  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**
    $S_L \| S_R \leftarrow \text{SPARKLE512}_{12}\big((\rho_1(S_L, \mathsf{pad}_{256}(M_{\ell_M - 1})) \oplus S_R \oplus \mathsf{Const}_M) \| (S_R \oplus \mathsf{Const}_M)\big)$
  else
    $S_L \| S_R \leftarrow \text{SPARKLE512}_{12}\big((\rho_1'(S_L, C_{\ell_M - 1}) \oplus S_R \oplus \mathsf{Const}_M) \| (S_R \oplus \mathsf{Const}_M)\big)$
  end if
end if
if $S_R \oplus K = T$ **then**
  return $(M_0 \| M_1 \| \ldots \| M_{\ell_M - 1})$
else
  return $\perp$
end if

B Origin of the Names

Sparkle is basically a Sparx instance with a wider block size and a fixed key, hence its name:

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

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]. Finally, 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].

C 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. 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.[13]

[13] 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.

```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;
  }
}
```