Alzette: A 64-Bit ARX-box: (feat. CRAX and TRAX)

Christof Beierle, Alex Biryukov, Luan Cardoso dos Santos, Johann Grossschädl, Léo Perrin, Aleksei Udovenko, Vesselin Velichkov, Qingju Wang

Alzette: a 64-bit ARX-box
(feat. CRAX and TRAX)

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

[1] Ruhr University Bochum, Germany (christof.beierle@rub.de)
[2] University of Luxembourg, Luxembourg (first-name.last-name@uni.lu)
[3] Inria, France (leo.perrin@inria.fr)
[4] CryptoExperts, France (aleksei@affine.group)
[5] University of Edinburgh, U.K. (vvelichk@ed.ac.uk)

sparklegrupp@googlegroups.com

Abstract. S-boxes are the only source of non-linearity in many symmetric primitives. While they are often defined as being functions operating on a small space, some recent designs propose the use of much larger ones (e.g., 32 bits). In this context, an S-box is then defined as a subfunction whose cryptographic properties can be estimated precisely.
We present a 64-bit ARX-based S-box called Alzette, which can be evaluated in constant time using only 12 instructions on modern CPUs. Its parallel application can also leverage vector (SIMD) instructions. One iteration of Alzette has differential and linear properties comparable to those of the AES S-box, and two are at least as secure as the AES super S-box. As the state size is much larger than the typical 4 or 8 bits, the study of the relevant cryptographic properties of Alzette is not trivial.
We further discuss how such wide S-boxes could be used to construct round functions of 64-, 128- and 256-bit (tweakable) block ciphers with good cryptographic properties that are guaranteed even in the related-tweak setting. We use these structures to design a very lightweight 64-bit block cipher (Crax) which outperforms SPECK-64/128 for short messages on micro-controllers, and a 256-bit tweakable block cipher (Trax) which can be used to obtain strong security guarantees against powerful adversaries (nonce misuse, quantum attacks).
Keywords: (tweakable) block cipher, related-tweak setting, long trail strategy, Alzette, MEDCP, MELCC

1 Introduction

It is well known that symmetric cryptographic primitives need to be non-linear. It is common to rely on so-called *S-boxes* to obtain this property. Typically

these are functions S mapping $\mathbb{F}_2^n$ to $\mathbb{F}_2^m$ for a value of n small enough that it is possible to specify S using its lookup table. They are applied in parallel to the whole state as part of the *round function* of the primitive.

This common definition of S-boxes is being challenged by the recent use of larger S-boxes in some designs. First, the designers of the hash function WHIRLWIND [5] used a 16-bit S-box based on the multiplicative inverse in the finite field $\mathbb{F}_{2^{16}}$. In this case, the intention of the implementers was not to use the 2^{17}-byte lookup table of the permutation but instead to rewrite the permutation using tower fields. More recently, large S-boxes have been proposed in Sparx [19] and in the NIST lightweight candidate Saturnin [15]. In the latter case, a 16-bit S-box is constructed using a classical Substitution-Permutation Network (SPN): four 4-bit S-boxes are applied to a 16-bit word in parallel, followed by an MDS matrix, and another application of the 4-bit S-box layer. While there is no closed formula for the differential and linear properties of such a structure (unlike for the multiplicative inverse used in WHIRLWIND), 16-bit remains small enough that a direct computation is possible.

This is not the case for the 32-bit S-box of Sparx. In this cipher, the S-box consists of an Addition, Rotation, XOR (ARX) network operating on two 16-bit branches, and it is key-dependent. Furthermore, while the properties of the S-box are usually sufficient[1] to prove that the cipher meets some security criteria, it is not the case for the *ARX-box* of Sparx. Indeed, in order to achieve the security goals by its designers (following the *long trail* security argument), it was necessary to study several "S-boxes", namely A, $A \circ A$, $A \circ A \circ A$, etc.

Another significant difference between the 32-bit ARX-box of Sparx and 16-bit S-boxes is the fact that it is *not* possible to evaluate its cryptographic properties directly because the complexity of the algorithms involved is usually proportional to 2^{2n}, where n is the block size. Thus, the authors of Sparx instead considered their ARX-box like a small block cipher and used techniques borrowed from block cipher analysis [14] to investigate their ARX-box.

Our Contribution. In this paper, we present a new 64-bit S-box called Alzette (pronounced [alzɛt]) that satisfies a similar scope statement to that of the Sparx ARX-box: it is also an ARX-based S-box, and we analyze both A and $A \circ A$. Alzette is parameterized by a constant $c \in \mathbb{F}_2^{32}$ and is defined for each such c as a permutation of $\mathbb{F}_2^{32} \times \mathbb{F}_2^{32}$. The algorithm evaluating this permutation is given in Algorithm 1 and depicted in Figure 1. Alzette has the following advantages:

- it relies on 32-bit rather than 16-bit operations, meaning that (according to [18, Sect. 5]) it is suitable for a larger number of architectures;
- it makes better use of barrel shift registers (when available) and has more efficient rotation constants (for platforms on which they have different costs);
- its differential and linear properties are superior to those of a scaled-up Sparx ARX-box;
- our analysis takes more attacks into account, and is confirmed experimentally whenever possible;

[1] Along with some conditions on the linear layer, in particular its branching number.

After providing a detailed design rationale of Alzette, we investigate its security against cryptanalytic attacks in more detail. Besides using state-of-the-art methods to conduct the analysis, we also developed new methods. In particular, to analyze the security against generalized integral attacks, we describe a new encoding of the bit-based division property [41] for modular addition.

Note that in some attack scenarios, the security of Alzette needs to be analyzed for the precise choice of round constants c used in the actual primitive. In this work, we provide experimental analysis for the round constants employed in the permutation Sparkle, submitted to the NIST lightweight cryptography standardization process [8]. However, our methods can easily be applied for an arbitrary choice of round constants.

Large parts of the experimental analysis have been carried out on the UL HPC cluster [42]. The source code for our experimental analysis can be found at https://github.com/cryptolu/sparkle.

We provide software implementations of Alzette on 8-bit AVR and 32-bit ARM processors. To summarize, Alzette can be executed in only 12 cycles on a 32-bit ARM Cortex-M3 and 122 cycles on an 8-bit AVR ATmega128 processor. Besides, the code size is low: respectively 24 and 176 bytes on those platforms.

Finally, we discuss the suitability of Alzette as a building block in cryptographic primitives. Since we already know how to use Alzette to design a cryptographic permutation, i.e., Sparkle, we show in this paper how it can be applied to design (tweakable) block ciphers operating on a variety of block lengths. In a nutshell, those ciphers use Alzette in a Feistel construction and interleave it with xoring the round keys. In a tweakable block cipher, the tweak will be xored only to half the state and only every second round. Similar to how the long-trail strategy was applied to take into account cancellations of differences within the absorption phase in a cryptographic sponge construction [8], we use the same technique to provide security arguments against related-tweak attacks, by taking cancellations of differences through tweak injection into account.

Besides describing this more general design idea, we provide two concrete cipher instances Crax and Trax.

Crax is a 64-bit block cipher that uses a 128-bit secret key. Since its key schedule is very simple and does not have to be precomputed, it is one of the fastest 64-bit lightweight block ciphers in software, beaten only for messages longer than 72 bytes by the NSA cipher Speck [6]. Due to this simple key schedule, it consumes lower RAM than Speck. While the family of tweakable block ciphers Skinny [9] can be considered as an academic alternative to the NSA cipher Simon [6] in terms of hardware efficiency, Crax can be seen as an academic alternative to Speck in terms of software efficiency.

Trax is a tweakable block cipher operating on a larger state of 256-bit blocks. It applies a 256-bit key and 128-bit tweak. To the best of our knowledge, the only other large tweakable block cipher is Threefish which was used as a building for the SHA-3 candidate Skein [33]. Unlike this cipher, Trax uses 32-bit words that are better suited for vectorized implementation as well as on micro-controllers. Another improvement of Trax over Threefish is the fact that we provide strong

bounds for the probability of all linear trails and all (related-tweak) differential trails. Because of its Substitution-Permutation Network structure, TRAX is indeed inherently easier to analyze. Such a large tweakable block cipher can provide robust authenticated encryption, meaning that it can retain a high security level even in case of nonce misuse or in the presence of quantum adversaries, as argued in [15]. The performance penalty of such guarantees can be minimized using vectorization and/or parallelism.

Algorithm 1 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)

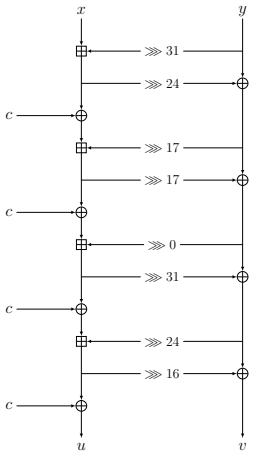

Fig. 1. The Alzette instance A_c.

Outline. The design process that we used to construct Alzette is explained in Section 2. In particular, we show that it offers resilience against a large variety of attacks. This analysis is confirmed experimentally in Section 3. We also discuss the efficiency of Alzette in Section 4. The discussion on the usage of Alzette as a building block, together with the specification of our (tweakable) block ciphers is given in Section 5.

Notation. By $\mathbb{F}_2$, we denote the finite field with two elements and by $\mathbb{F}_2^n$ the set of bitstrings of length n. We denote the set $\{0, 1, \ldots, n-1\}$ by $\mathbb{Z}_n$. We use $+$ to denote the addition modulo 2^{32} and $\oplus$ to denote the XOR of two bitstrings of the same size. The symbol & denotes the bit-wise AND. Further, by $x \ggg r$, we denote the cyclic rotation of the 32-bit word x to the right by the offset r.

Let E be a key alternating block cipher with r rounds, and round function R. In a differential attack [12] against E_k, an attacker exploits differences δ and Δ such that the probability that $E_k(x \oplus \delta) \oplus E_k(x) = \Delta$ is significantly higher than 2^{-n} (for an n-bit block cipher). For typical values of n (64, 128 or 256) the exact computation of this probability is infeasible. Instead, the common practise is to approximate this quantity by the maximum probability of a *differential trail/characteristic* averaged over all round keys. A differential trail is a

sequence of differences $\{\delta_0, \delta_1, ..., \delta_r\}$ that specifies not only the input and output differences to the block cipher, but also the intermediate differences between the rounds such that $R(\delta_i \oplus x) \oplus R(x) = \delta_{i+1}$. The approximated probability (averaged over all round keys) is derived as the product of the probabilities of the transitions occurring in each round[2]. The maximum probability (across all trails) computed in this way is denoted *Maximum Expected Differential Characteristic Probability (MEDCP)*. An upper bound on the MEDCP is an approximation of the maximum differential probability and is called a *differential bound*.

By analogy to the differential case, for linear attacks [31] the aim is to find masks α and β such that $\beta \cdot E_k(x) = \alpha \cdot x + f(k)$, where "$\cdot$" denotes the usual scalar product over $\mathbb{F}_2^n$ and where f is a function of the key bits. In practice, we look for a sequence of input, output and intermediate masks $\{\alpha_0, ..., \alpha_r\}$ called a *linear trail/characteristic* that has high absolute correlation, where $\alpha_{i+1} \cdot R(x) = \alpha_i \cdot x + f_i(k)$. Analogously to MEDCP and the differential bound, in the linear case we define a *Maximum Expected Linear Characteristic Correlation (MELCC)* and a *linear bound*.

2 The Design of Alzette

We now present both the design process and the main properties of Alzette. These are verified experimentally later in Section 3, and summarized in Section 3.6.

2.1 Block and Word Sizes

Our S-box should be efficient on a wide variety of platforms, while allowing a practical analysis of its relevant cryptographic properties. What would be the best word and block sizes in this context?

Word size. In Sparx, the S-box operates on 32 bits, which are split into two 16-bit words. This word size allows a computationally cheap analysis of its cryptographic properties while facilitating efficient implementations on 8 and 16-bit micro-controllers. However, 16-bit words hamper performance on 32-bit platforms, simply because only half of their 32-bit registers and datapath can be used. The same holds when 16-bit operations are executed on a 64-bit processor. Furthermore, 16-bit operations can also incur a performance penalty on 8-bit micro-controllers; for example, rotating two 16-bit operands by n bits on an 8-bit AVR device is usually slower than rotating a single 32-bit operand by n bits (see e.g. [17, Appendix A, B, C] for details).

While 16-bit words are suboptimal because they are too small, it can also be argued that 64-bit word are too large. To establish why, we have to separately discuss the performance of 64-bit operations on 8/16/32-bit micro-controllers and on 64-bit processors. We start with three arguments for why 64-bit operations may not be a good choice on small micro-controllers.

[2] Under the *Markov assumption* which allows to treat the rounds as independent from each other.

1. 32-bit ARM micro-controllers allow one to perform a rotation "for free" since it can be executed together with another arithmetic/logical instruction.[3] Still, a 32-bit ARM processor can only perform rotations of 32-bit operands for free, but not rotations of 64-bit words.
2. As discussed later, we will use word-wise modular additions. Some 32-bit architectures, most notably RISC-V and MIPS32, do not have an add-with-carry instruction. Adding two 64-bit operands on these platforms requires to first add the lower 32-bit parts of the operands and then compare the 32-bit sum with any of the operands to find out whether an overflow happened (i.e. to obtain a carry bit). Then, the two upper 32-bit words are added up together with the carry bit. A 64-bit addition requires at least four instructions (i.e. four cycles) on these platforms, whereas two 32-bit additions take only two instructions (i.e. two cycles).
3. Compilers for 8 and 16-bit micro-controllers are notoriously bad at handling 64-bit words, especially rotations of 64-bit words. The reason is simple: outside of cryptography, 64-bit words are of little to no use on an 8- or 16-bit platform, and therefore compiler designers have no incentive to optimize 64-bit operations.

A word size of 64 bits is naturally a good choice for 64-bit processors. For example, the authors of [21] established that SHA512 (which operates in 64-bit words) reaches much higher throughput on 64-bit Intel processors than SHA256 (operating on 32-bit words). However, this does not necessarily imply that ARX designs using 32-bit words are inferior to 64-bit variants on 64-bit processors. This can be justified with the fact that the best way to implement an ARX cipher on a 64-bit Intel or a 64-bit ARM processor is to use the vector (SIMD) extensions they provide, e.g. Intel SSE, AVX or ARM NEON. Most high-end 64-bit processors have such vector instruction sets, and all of them can execute additions, rotations and XORs on 32-bit words. The fact that a 32-bit word size allows peak performance on 64-bit processors was already used for instance by the designers of Gimli [11].

As a consequence, we chose to design an S-box that operates on 32-bit words as those offer the best performances across the board.

Block size. Our S-box could a priori operate on any block size that is a multiple of 32. However, two criteria significantly narrow down the design space.

First, we need to be able to investigate the cryptographic properties of our S-box. We are not aware of any efficient combination of simple operations (AND, addition, rotation, XOR, etc.) on a single word that would allow us to give strong bounds on the differential and linear probabilities. On the other hand, computational technique that find such bounds tend to be less efficient if the state size is large as it implies a greater number of potential branches to explore in a tree. Our ability to find bounds thus imposes a number of words which is at least equal to 2 and as small as possible.

[3] We exploit this property to design Alzette, as explained in Section 2.2.

Second, in order to use vector instruction sets to their fullest extent, it is better to have a larger number of S-boxes that can be applied in parallel in each call to the round function. On smaller micro-controllers, limiting the block size makes it easier for implementers to keep one full S-box state (or maybe even several full S-box states) in the register file, thereby reducing the number of memory accesses. Finally, in order to build primitives with a small state size, it is necessary that the S-box size is at most equal to said state size. However, as mentioned before, it makes sense to aim for the smallest possible number of branches (and, consequently, a large number of S-boxes) to leverage SIMD-style parallelism.

Because of these requirements, we settled for the use of two words. Given that our discussion above set a 32-bit word size, our S-box operates on 64 bits.

2.2 Round Structure and Number of Rounds

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

In particular, we aimed for designing an ARX-box consisting of the composition of t rounds of the form

$$T_i : \begin{cases} \mathbb{F}_2^{32} \times \mathbb{F}_2^{32} & \to \mathbb{F}_2^{32} \times \mathbb{F}_2^{32} \\ (x, y) & \mapsto \big(x + (y \lll r_i),\ y \oplus \big((x + (y \lll r_i)) \lll s_i \big)\big) \oplus (\gamma_i^L, \gamma_j^R) \ , \end{cases}$$

where i-th round is defined by the rotation amounts $(r_i, s_i) \in \mathbb{Z}_{32} \times \mathbb{Z}_{32}$ and the round constant $(\gamma_i^L, \gamma_i^R) \in \mathbb{F}_2^{32} \times \mathbb{F}_2^{32}$. It is computed in three steps: $x \leftarrow x + (y \lll r_i)$, $y \leftarrow y \oplus (x \lll s_i)$, and finally $(x, y) \leftarrow (x \oplus \gamma_i^L, y \oplus \gamma_i^R)$.

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

2.3 Criteria for Choosing the Rotation Amounts

We aimed for choosing the rotations (r_i, s_i) in Alzette in a way that maximizes security and efficiency. For efficiency reasons, we want to minimize the *cost* of the rotations, where we use the cost metric as given in Table 1. While each rotation has the same cost in 32-bit ARM processors (i.e., 0 because rotation is for free on top of XOR, resp., AND), we further aimed for minimizing the cost with regard to 8-bit and 16-bit architectures. Therefore, we restricted ourselves to rotations from the set $\{0, 1, 7, 8, 9, 15, 16, 17, 23, 24, 25, 31\}$, as those are the most efficient when implemented on 8 and 16-bit micro-controllers. We define the *cost* of a collection of rotation amounts (that is needed to define all the rounds of an ARX-box) as the sum of the costs of its contained rotations.

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

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

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

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

– The 8-round linear bound ($-\log_2$) of the candidate ARX-box should be at least 17.

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

We tested *all* rotation sets with a cost below 12 for the above conditions. None of those fulfilled the above criteria. For a cost below 15, we found the ARX-box with the following rotations:

$$(r_0, r_1, r_2, r_3, s_0, s_1, s_2, s_3) = (31, 17, 0, 24, 24, 17, 31, 16) \ .$$

This rotation set fulfills all the criteria. The differential and linear bounds for the respective ARX-box are summarized in Table 2.

Table 2. Differential and linear bounds for several rotation parameters. For each offset, the first line shows the differential bound and the second shows the linear one. The value set in parenthesis is the maximum absolute correlation of the linear hull taking clustering into account (see Section 3.2). The bounds [14,20,28,29,30] for SPECK are given for comparison.

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

2.4 On the Round Constants

The purpose of round constant additions, i.e., the XORs with γ_i^L, γ_i^R in the general ARX-box structure, is to ensure some independence between the rounds. They also break additive patterns that could arise on the left branch due to the chain of modular addition it would have without said constant additions. Perhaps even more importantly, they should also ensure that the Alzette instances called in parallel are different from one another to avoid symmetries.

For efficiency reasons, we decided to use the same round constant in every round of the ARX-box, i.e., $\forall i : \gamma_i^L = c$. As the rounds themselves are different

from one another, we do not rely on γ_i^L or γ_i^R to prevent slide-style patterns. Thus, using the same constant in each round is not a problem. Moreover, we chose $\gamma_i^R = 0$ for all i. It is important to note that the experimental verification of the differential probabilities and absolute linear correlations we conducted (see Sections 3.1 and 3.2 respectively) did not lead to significant differences when changing to a more complex round constant schedule. In other words, even for random choices of all γ_i^L and γ_i^R, we did not observe significantly different results that would justify the use of a more complex constant schedule (which would of course lead to worse efficiency in the implementation).

The analysis provided in the next section is dependent on the actual choice of round constants c. We conducted this analysis for the constants of Sparkle:

$$\begin{aligned} &c_0 = \texttt{b7e15162}, c_1 = \texttt{bf715880}, c_2 = \texttt{38b4da56}, c_3 = \texttt{324e7738}, \\ &c_4 = \texttt{bb1185eb}, c_5 = \texttt{4f7c7b57}, c_6 = \texttt{cfbfa1c8}, c_7 = \texttt{c2b3293d} \,. \end{aligned} \tag{1}$$

3 Analysis of Alzette

In this section, we study cryptographic properties of the ARX-box Alzette. The analysis is done for the round constants used in Sparkle, except for analysis of differential/linear characteristic bounds and division property propagation, which are independent of the choice of the constants. All described methods can easily be applied to arbitrary choices of constants.

3.1 On the Differential Properties

Bounding the Maximum Expected Differential Trail Probability. We used the Algorithm 1 in [14] and adapted it to our round structure to compute the bounds on the maximum expected differential trail probabilities of the ARX-boxes which use the constants given in Equation (1). The algorithm is basically a refined variant of Matsui's well-known branch and bound algorithm [32]. While the latter has been originally proposed for ciphers that have S-boxes (in particular the DES), the former is targeted at ARX-based designs that use modular addition, rather than an S-box, as a source of non-linearity.

Algorithm 1 [14] exploits the differential properties of modular addition to efficiently search for characteristics in a bitwise manner. Upon termination, it outputs a trail (characteristic) with the maximum expected differential trail probability (MEDCP). For Alzette, we obtain such trails for up to six rounds, where the 6-round bound is 2^{-18}. We further collected all trails corresponding to the maximum expected differential probability for 4 and 5 rounds and experimentally checked the actual probabilities of the differentials (for the constants used in Sparkle), see below.

Note that for 7 and 8 rounds, we could not get a tight bound due to the high complexity of the search. In other words, the algorithm did not terminate in reasonable time. 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. We evaluated similar bounds for up to 12 rounds.

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

For all of the 8 Alzette instances corresponding to the constants in Equation (1), we conducted experiments in order to see if the expected maximum differential trail probabilities derived by Matsui's search are close to the actual differential probabilities of the fixed ARX-boxes. Our results are as follows.

By Matsui's search we found 7 differential trails for Alzette[4] that correspond to the maximum expected differential trail probability of 2^{-6} for 4 rounds, see Table 3. For any Alzette instance A_{c_i} and any such trails with input difference α and output difference β, we experimentally computed the actual differential probability of the differential $\alpha \to \beta$ by

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

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

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

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

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

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

Table 3. The input and output differences α, β (in hex) of all differential trails over Alzette corresponding to maximum expected differential trail probability $p = 2^{-6}$ and $p = 2^{-10}$ for four and five rounds, respectively.

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

3.2 On the Linear Properties

Bounding Maximum Expected Absolute Linear Trail Correlation. We used the Mixed-Integer Linear Programming approach described in [20] and the Boolean satisfiability problem (SAT) approach in [28] in order to get bounds on the maximum expected absolute linear trail correlation. It was feasible to get tight bounds even for 8 rounds, where the 8-round bound of our final choice for Alzette is 2^{-17}. We were able to collect all linear trails that correspond to the maximum expected absolute linear trail correlation for 4 up to to 8 rounds and experimentally checked the actual correlations of the corresponding linear approximations for the Alzette instances using the constants in Equation (1), see below.

Experiments on the Fixed-Key Linear Correlations. Similarly as for the case of differentials, for all of the 8 Alzette instances used in Sparkle, we conducted experiments in order to see whether the maximum expected absolute linear trail correlations derived by MILP and presented in Table 2 are close to the actual absolute correlations of the linear approximations over the fixed Alzette instances. Our results are as follows, and presented in Table 12 in Appendix A.

For a full Alzette instance, there are 4 trails with a maximum expected absolute trail correlation of 2^{-2}. For all of the eight Alzette instances, the actual absolute correlations are very close to the theoretical values and we did not observe any clustering. For more than four rounds (i.e., one full instance plus additional rounds), we again checked all combinations of ARX-boxes that do not get a step counter in Sparkle. For five rounds, there are 16 trails with a maximum expected absolute trail correlation of 2^{-5}. In our experiments, we can observe a slight clustering. The observed absolute correlations based on 2^{24} samples can also be found in Table 12. The minimum and maximum refers to the minimum, resp., maximum observed absolute correlations over all the combinations of Alzette instances that do not get a step counter, similar as tested for differentials. In fact, we chose the round constants c_i of Sparkle such that, for all combinations of Alzette that occur over the linear layer, the linear hull effect

is to our favor, i.e., the actual correlation tends to be *lower* than the theoretical value.[5]

This tendency also holds for the correlations over six rounds. There are 48 trails with a maximum expected absolute linear trail correlation of 2^{-8}. The results of our experiments for 2^{28} random samples are shown in Table 13 in Appendix A.

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

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

Overall, our correlation estimates based on linear trails seem to closely approximate the actual absolute correlations since our estimate is only $2^{1.21}$ times lower than the actual absolute correlation.

3.3 On the Algebraic Properties

Integral cryptanalysis exploits low algebraic degree or a more fine-grained algebraic degeneracy of the cryptographic primitive under attack. An *integral distinguisher* defines an input set X such that the analyzed function sums to zero over this set (at least in some bits) for any value of the secret key involved. In the case of a keyless permutation, such as an ARX-box, such distinguishers are trivial to find and are meaningless. However, an analysis of the growth of the algebraic degree (and the evolution of the algebraic structure in general) provides a useful information about the permutation. When the permutation is plugged into, for example, a block cipher, this information directly translates into information about integral distinguishers.

Division property is a technique introduced by Todo [39] to find *integral characteristics*. Originally, it was applied to substitution-permutation networks and Feistel networks. Later, *bit-based division property* was proposed by Todo and Morii [41] and applied to the Simon block cipher with 32-bit blocks. Due to the high computation complexity of the search algorithm, it is infeasible to apply the technique to ciphers with larger block sizes. However, Xiang *et al.* [43] discovered that the bit-based division property propagation can be efficiently encoded as *an mixed-integer linear programming* instance (MILP), and, surprisingly, can be solved on practice using modern optimization software (Gurobi Optimizer [22])

[5] The constants in Sparkle were derived from the fractional digits of e, excluding some blocks. For the excluded blocks, the actual absolute correlations are slightly higher than the theoretical bound, but all smaller than 2^{-8}.

for practically all known block ciphers. Sun *et al.* [38] described a way to encode the modular addition operation using MILP inequalities, extending the framework to ARX-based primitives.

We briefly recall the MILP-aided bit-based division property framework.

Definition 1 (Block-Based Division Property). *Let n be an integer and let X be a set of n-bit vectors. Let k be an integer, $0 \leq k \leq n$. The set S satisfies division property $\mathcal{D}_k^n$ if and only if for all $u \in \mathbb{F}_2^n$ with $wt(u) < k$, we have $\bigoplus_{x \in X} x^u = 0$, where x^u is a shorthand for $x_0^{u_0} \dots x_{n-1}^{u_{n-1}}$.*

Definition 2 (Bit-Based Division Property). *Let n be an integer and let X, K be two sets of n-bit vectors, $0 \notin K$. The set X satisfies division property $\mathcal{D}_K$ if and only if for all $u \in \mathbb{F}_2^n$ such that $u \prec k$ for all $k \in K$*

$$\bigoplus_{x \in X} x^u = 0 \,,$$

where $u \prec k$ if and only if $u \neq k$ and $u_i \leq k_i$ for all $i, 0 \leq i < n$.

Remark 1. For any $j, 0 \leq j < n$, if the j-th unit vector does not belong to K, then $\bigoplus_{x \in S} x_j = 0$, and the j-th bit of the set X is said to be *balanced*.

The division property can be used to find integral characteristics in the following way. First, a particular initial division property $\mathcal{D}_K$ is chosen. Typically, K consists of a single vector of Hamming weight d for some integer d. The minimum size of a set satisfying $\mathcal{D}_K$ is then 2^d (for instance, a particular *cube* of dimension d). Then, the division property is propagated through the rounds of the analyzed primitive using rules specific for each bit operation (copy, xor, and); for propagation through an S-Box, a special propagation table is generated. Finally, Remark 1 is used to check if an integral distinguisher is found.

For further information on division property propagation and its encoding using MILP inequalities, we refer to [43]. However, we describe briefly a new technique for encoding division property propagation through the modular addition. Our technique is simpler and more compact than the one proposed by Sun *et al.* [38].

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

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

Consider an addition of two n-bit words $a, b \in \mathbb{F}_2^n$ and let $y = a \boxplus b \mod 2^n$ (recall that a_0 denotes the most significant bit of a, a_{n-1} denotes the least significant bit of a, etc.). Define *carry* bits $c_i, 0 \leq i < n$ as follows: $c_{n-1} = 0$ and

$c_i = \mathrm{Maj}(a_{i+1}, b_{i+1}, c_{i+1})$ for $-1 \leq i < n-1$, where Maj is the 3-bit *majority* function. Then it is easy to verify that $y_i = a_i \oplus b_i \oplus c_i$ for all $0 \leq i < n$. Full modular addition can be computed sequentially from $i = n-1$ to $i = 0$. Let $f : \mathbb{F}_2^3 \to \mathbb{F}_2^2$ be such that $f(a, b, c) = (\mathrm{Maj}(a, b, c), a \oplus b \oplus c)$, then we can write

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

for all $0 \leq i < n$. The lookup table of f is given in Table 4. Note that no bits are copied in the sequential computation process. It follows that the division property propagation can be encoded directly by encoding n sequential applications of f (using the S-Box encoding methods by Xiang *et al.* [43]). Finally, an additional constraint is needed to ensure that the resulting division property is not active in the bit c_{-1}.

The division property propagation table is given in Table 5. This table can be characterized by the two following integer inequalities:

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

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

Table 4. Look-up table of f.

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

Table 5. Division property propagation table of f.

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

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

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

We used this representation together with the Boolector SMT-solver [34] (version 3.1.0) to verify our results.

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

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

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

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

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

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

Table 6. The upper bounds on the minimum and maximum degree of the coordinates of Alzette and its inverse.

Rounds	1	2	3	4	Inverse rounds	1	2	3	4
min	1	10	42	63	min	1	2	32	46
max	32	62	63	63	max	32	62	63	63

though these are just upper bounds, we expect that they are close to the actual values, as the division property was shown to be rather precise [41]. Thus, the Alzette structure may have full degree in all its coordinates, but the inverse of an Alzette instance has a coordinate of degree 46.

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

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

bits from the right), and in the output 16 bits are balanced:

input active bits:
`11111111111111111111111111111111,11111111111101111111111111111111`,
balanced bits after 6-round ARX-box (denoted by `B`):
`????????????????????????BBBBBBBB,??????????BBBBBBBB??????????????`.

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

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

input active bits:
`11111111111111111111111111101111,11111111111111111111111111111111`,
balanced bits after 5-round ARX-box inverse:
`????????????????????????????????B,???????BBBBBBBBB??????????????`.

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

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

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

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

3.4 Invariant Subspaces

Invariant subspace attacks were considered in [27]. For the round constants used in Sparkle, using a similar "to and fro" method from [35,13], we searched for an affine subspace that is mapped by an Alzette instance A_{c_i} to a (possibly different) affine subspace of the same dimension. We could not find any such subspace of nontrivial dimension.

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

such that $A(l, r) = (l', r')$, it holds that

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

This equation defines a subspace trail of constant dimension 59.

3.5 Nonlinear Invariants

Nonlinear invariant attacks were considered recently in [40] to attack lightweight primitives. For the round constants used in Sparkle, using linear algebra, we experimentally verified that for any ARX-box A_{c_i} and any non-constant Boolean function f of degree at most 2, the compositions $f \circ A_{c_i}$ and $f \circ A_{c_i}^{-1}$ have degree at least 10:

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

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

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

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

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

3.6 Summary of the Properties of Alzette

Our experimental results validate our theoretical analysis of the properties of Alzette: in practice, the differential and linear trail probabilities (resp., absolute correlations) are as predicted. In the case of differential probabilities, the clustering is minimal. While it is not quite negligible in the linear case, our estimates remain very close to the quantities we measured experimentally.

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

While the two components have utterly different structures, Alzette has similar properties to one round of AES and the double iteration of Alzette to the AES super-S-box (see Table 7). The bounds for the (double) ARX-box come from Table 2. For the AES, the bounds for a single rounds are derived from the properties of its S-box, so its maximum differential probability is $4/256 = 2^{-6}$ and its maximum absolute linear correlation is 2^{-3}. For two rounds, we raise the quantities of the S-box to the power 5 because the branching number of the MixColumn operation is 5,

Table 7. A comparison of the properties of Alzette with those of the AES with a fixed key. MEDCP denotes the maximum expected differential trail probability and MELCC denotes the maximum expected absolute linear trail correlation.

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

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

4 Implementation Aspects

4.1 Software Implementations

Alzette 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 $z \leftarrow x <op> (y \lll \ell)$ can be executed with a single instruction in a single clock cycle, irrespective of ℓ.

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 a primitive calling Alzette.

The consistency of operations allows one to either focus on small code size (by implementing the parallel Alzette instances in a substitution layer in a loop), or on architectures with more registers, execute two or more instances 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. Due to the layout of Alzette, an 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.

Table 8 summarizes the execution time and code size of Alzette on an 8-bit AVR and a 32-bit ARM Cortex-M3 micro-controller. The assembler implementation of Alzette for the latter architecture consists of 12 instructions (see Appendix C), which take 12 clock cycles to execute. The actual code size of Alzette may be less than 48 bytes since the Cortex-M3 supports Thumb2, which means some simple instructions can be only 16 bits long. However, whether an

Table 8. Execution time (in clock cycles) and codes size (in bytes) of Alzette.

Platform	Execution time	Code size
8-bit AVR ATmega128	78	156
32-bit ARM Cortex-M3	12	24

instruction is 16 or 32 bits long depends, among other things, on the register allocation. Our ARM implementation assumes that the two 32-bit branches of Alzette and the round constant are already in registers and not in memory, which is a reasonable assumption since the register file of a Cortex-M3 is big enough to accommodate a few instances of Alzette together with a few round constants.

The situation is a bit different for 8-bit AVR. The arithmetic/logical operations of Alzette amount to 78 instructions altogether, each of which executes in a single cycle, i.e. 78 clock cycles in total. Each of the used instructions has a length of 2 bytes, yielding a code size of 156 bytes. However, in contrast to ARM, we can not take it for granted that the whole state of a cipher fits into the register file of an AVR micro-controller, which means the load and store operations should be considered when evaluating the execution time. Loading a byte from RAM takes 2 cycles, while loading a byte from flash (e.g. for the round constants) requires 3 cycles. Storing a byte in RAM takes also 2 cycles. Consequently, when taking all loads/stores into account (including the loading of a round constant from flash), the execution time increases from 78 to 122 cycles and the code size from 156 to 196 bytes.

4.2 Hardware Implementations

A hardware implementation can, for example, use a 32-bit ALU that is 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.

5 Alzette as a Building Block

Alzette is at the core of two families of lightweight algorithms that are among the second round candidates of the NIST lightweight cryptography standardization process, namely the hash functions Esch and the authenticated ciphers with associated data Schwaemm (submission Sparkle [8]). In this section, we show that it can also be used to easily construct block ciphers. This approach is flexible: combining Alzette with simple linear layers, we can simply build step functions operating on 64-, 128- and 256-bit blocks. We explain this approach and analyze the security of its result in Section 5.1. Specific instances are then given in Section 5.2, namely the 64-bit lightweight block cipher Crax, and the 256-bit tweakable block cipher Trax.

5.1 Skeletons for a Family of (Tweakable) Block Ciphers

Our approach relies on the long trail strategy pioneered by the designers of Sparx [19], and which was then used to build sLiSCP [3], sLiSCP-light [4] as well as the NIST lightweight candidates using them (SPIX [2], SPOC [1], Sparkle [8]). Provided that the round function allows its use, this method provides a simple algorithm for bounding the probability of differential and linear trails. To achieve this, we loop over all possible truncated trails, and bound the probability of all differential (resp. linear) trails that conform to the truncated trail using the differential (resp. linear) bounds of the employed S-box, including those for multiple iterations when relevant. In all the algorithms listed above, variants of the Feistel structure have been applied because such round functions lend themselves well to such an analysis.

It is simple to adapt this framework to the design of Alzette-based block ciphers. Furthermore, the structure of a long trail argument allows for an efficient algorithm bounding the probability of *related-tweak* differentials.[6] Indeed, in our case, the S-box used is 64 bit wide. Thus, the number of bits needed to describe a truncated differential in a given internal state is very small, only 4 suffice for a block size of 256 bits. Besides, the use of a Feistel structure implies that half of these bits are mere copies of the ones in the previous round. As a consequence, the total number of truncated trails that must be considered is low.

It also implies that the impact of a tweak difference is manageable: if the tweak difference activates a previously inactive S-box then its presence does

[6] Note that, in a related-tweak differential, we allow non-zero input differences not only in the plaintext, but also in the tweak value. This is because the attacker can choose the tweak, i.e., he has access to an encryption oracle for the cipher instantiated with a tweak T and a (random) key K and to an encryption oracle for a cipher instantiated with tweak $T \oplus \Delta$ and key K, where Δ can be freely chosen.

not increase the number of truncated trails. On the other hand, a possible cancellation merely multiplies the number of possible trails by 2. An algorithm enumerating all related-tweak truncated trails such that the probability of all differential trails that conform to them is below a given threshold, is therefore easy to write and is efficient. In fact, our straight-forward Python implementation returned all the results needed for this paper in a matter of seconds at worst. Large S-boxes such as Alzette are therefore very convenient building blocks to construct tweakable block ciphers with strong security arguments.

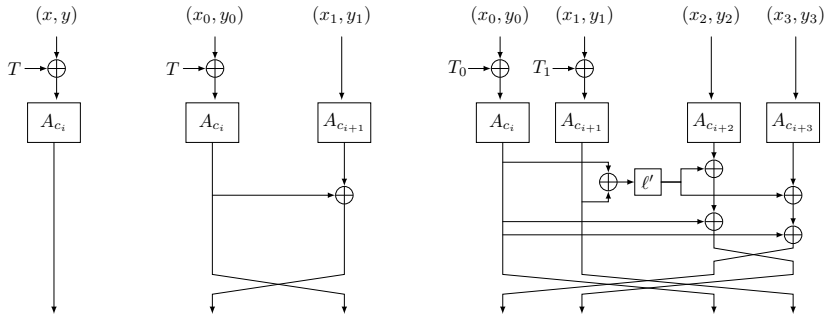

Fig. 2. The round functions of Trax-S, Trax-M, and Trax-L, respectively. $\ell'(z_1, z_2, z_3, z_4) = (z_4, z_3 \oplus z_4, z_2, z_1 \oplus z_2)$, where z_i are 16-bit words. The tweak is added only in odd steps.

Below, we present three Alzette-based (tweakable) block cipher structures for which we provide upper bounds on the probability of the best differential trail in both the single-key and the related-tweak setting. Of course, we also investigate other attacks. The "S", "M" and "L" versions operate on 64, 128 and 256 bits respectively and their round functions are depicted in Figure 2 (pseudo-code is provided in Appendix B). Their properties are summarized in Table 9:

- $r_e(c)$ rounds are needed to prevent the existence of known single-key distinguishers with a data complexity upper bounded by 2^c in total,
- $r_e^T(c)$ rounds are needed to prevent the existence of known related-tweak distinguishers with a data complexity upper bounded by 2^c in total (possibly spread across multiple tweak values), and
- r_d rounds are needed in order for all the bits of the state to depend on all the bits of the key.

For example, if the best single-key differential trail with a probability above 2^{-n} covers r rounds, then $r_e \geq r + 1$. It is assumed that n-bit subkeys are used.

It is assumed that there is no tweak schedule, i.e. that the tweak is simply XORed in the same part of the state each time it is added. As discussed below, we found that the security level was higher when this addition occurred every second step. The motivation for this simple tweak-schedule is simple: the tweak is

expected to change far more often than the key, so using a trivial tweak-schedule will improve the performances of our algorithms.

Of course, we can set the tweak to a constant (e.g. 0) and obtain a tweakless "regular" block cipher.[7] For the skeleton structures, we do not specify key schedules and leave it to cipher designers to come up with appropriate ones for their use cases. Related-key and related-tweak security will of course depend on the specifics of the key schedule chosen. We present concrete ciphers using these structures in Section 5.2 (along with their key schedules). Our best distinguishers against the various versions of our step function are summarized in Table 10. Details about related-tweak integral distinguishers are given in Appendix E. In order to evaluate r_e^T for such distinguishers, we used best integral distinguishers for forward and backward direction (ignoring the difference in the input pattern), summed the rounds and increased/rounded up by one step. The distinguishers use full tweak and all but one plaintext bits, and have data complexity $2^{n+|T|-1}$. Though it is possible that some active bits are not necessary for the distinguisher and can be set inactive, we expect that such distinguishers provide very loose bounds.

Table 9. The properties of the different (tweakable) round functions. The results include the full-data RT integral distinguishers.

Version	n	$\|T\|$	r_d	r_e	$r_e^T(n)$	$r_e^T(n/2)$
S	64	64	2	5	8	8
M	128	64	2	7	11	11
L	256	128	3	10	16	14

Table 10. The number of steps $r_e(i)$ needed for the S, M and L step functions to prevent various distinguishers with a data complexity of at most 2^i. "RT" stands for "related-tweak" where the tweak is added in every odd step. As $r_e(n/2) \leq r_e(n)$, we use the latter if the former is not known and use "†". For comparison, we give $r_e(i)$ for the `AES` using thatit achieves 25 active S-boxes in any non-trivial 4-round (differential or linear) trail and plugging in the bounds for its S-box provided in Table 7.

	Distinguisher	Differential	Linear	Imp. diff.	RT differential	RT integral
S	$r_e(n)$	4	5	2	8	8
	$r_e(n/2)$	2	3	$2^\dagger$	4	$8^\dagger$
M	$r_e(n)$	7	7	4	11	11
	$r_e(n/2)$	4	4	4	6	$11^\dagger$
`AES-128`	$r_e(n)$	4	4	5	–	–
	$r_e(n/2)$	4	4	$5^\dagger$	–	–
L	$r_e(n)$	10	10	4	16	14
	$r_e(n/2)$	5	6	4	9	$14^\dagger$

[7] We do *not* consider related-cipher attacks between the obtained block cipher and the corresponding tweakable block cipher.

The S Version. It operates on 64 bits, meaning that it simply consists in iterating Alzette, interleaving it with key additions. The tweak is XORed every second step as it allows to ensure that at least one double Alzette is active during 4 steps. Thus, 8 steps are sufficient to prevent related-tweak differential distinguishers with a data complexity of 2^{64}. If we remove the tweak then we need 4 steps to argue the absence of differential distinguishers.

We start adding the tweak at the beginning of step 1 and not step 0 as it could otherwise trivially be cancelled out with chosen plaintexts.

As we saw in Section 3.2, linear distinguishers are in practice less predictable than differential ones. In particular, they exhibit some key-sensitivity that we did not observe in the differential case. As our bound for 4 steps is at the edge of being exploitable (2^{-34}), a small key-dependent deviation may allow 4-step distinguishers. As a consequence, we consider that 5 steps are needed to prevent linear distinguishers. Note that, allowing related tweaks does not give an advantage when looking for linear distinguisher, as established by Kranz et al. [25].

The security against integral attacks and other attacks that would exploit a slow diffusion (like impossible differential attacks) also follows directly from our analysis of Alzette: our best integral distinguisher relies on the bit-based division property and covers only 6 rounds of Alzette, i.e. 1.5 steps. Extending it backwards, we can obtain at most an 11-round zero-sum distinguisher, i.e. one that covers 2.75 steps. Thus, 3 steps are sufficient to prevent them. Since we have full diffusion in one step, there cannot be an impossible differential found via a miss-in-the-middle that covers 2 steps.

Assuming that the key schedule uses statistically independent key bits in even and odd steps, we need only $r_d = 2$ steps to ensure that all bits depend (although possibly weakly) on all key bits. This result, along with all the distinguishers we investigated for this step function, are summarized in Table 10.

The M Version. In order to operate on 128 bits, we use a simple Feistel round as the linear layer that maps (x, y) to $(y \oplus x, x)$. This structure ensures long trails. To further foster the existence of long trails, we only XOR the tweak on half of the state, namely at the input of the Alzette instance which is always doubled due to the structure of the linear layer.

We have found using our long trail argument implementation that the best frequency for adding a tweak corresponds to an addition every second round (as for the S version). A smaller or larger number of steps between tweak additions would lead to worse differential bounds. As in the S version, we start adding the tweak at the beginning of step 1.

A long trail argument shows that differential and linear distinguishers become infeasible when the number of steps is at least equal to 7. Unlike in the S version, trail clustering is less of a concern here. Indeed, we observed the clustering within one Alzette to be minimal, and unlike in the S version, the linear masks are constrained in each step by the presence of the linear layer. It is not sufficient for the input and output masks of a double Alzette to be identical: in order to leverage clustering, we now need that the mask at the end of the first Alzette call is the same in all trails as well.

In the related-tweak setting, there could exist differential trail covering more than 7 steps with usable probabilities but none covering 11 steps (or more). If we restrict ourselves to attack with a data complexity at most equal to 2^{64} then no useful related-tweak trail can cover more than 6 steps.

This step function employs a Feistel structure with a bijective Feistel function but the well known 5-round impossible differential identified by Knudsen [24] cannot be used here. Indeed, our non-linear permutation (Alzette) is applied on both branches in each round, thus breaking the pattern used by this distinguisher. In fact, the best impossible differential we can find only covers 4 steps: the probability of the transition $(0, \delta) \rightsquigarrow (\Delta, \Delta)$ is equal to 0 for any non-zero 64-bit differences δ and Δ. It needs about 2^{32} chosen plaintexts to be exploited.

Since the key is of the same size as the block, the number of rounds needed for diffusion of the key material is the number of rounds needed all state bits to impact the whole state. In this case, it is $r_d = 2$.

The L Version. This round function operates on 256-bit using 4 Alzette instances in parallel. The round key is added in the full state. The best frequency for adding the tweak is every second step, for the same reason as for the M version: changing this frequency leads to worse differential bounds in the related-tweak setting.

This round function is similar to the one of Sparkle: a lot of the cryptanalysis performed for this algorithm directly carries over (see [8]). In particular, the type of attacks for which we need the largest number of steps to prevent the existence of distinguishers is indeed the linear one in the single-tweak setting.

As for the M version, the number of rounds needed for diffusion of the key is the number needed for all state bits to impact the whole state. Here, it is $r_d = 3$.

5.2 Recommended Instances

Choosing the Number of Steps. In order to evaluate the number of steps needed to build a secure cipher, we observed that attacks against block ciphers are usually constructed using a specific distinguisher against a round-reduced version of the algorithm. Then, rounds are added at the top and at the bottom using key guesses. As a consequence, we used the following heuristic.

Heuristic (Number of rounds). *Suppose that a block cipher round function is such that:*

- r_e *rounds are needed to prevent the existence of known (and relevant) distinguishers, and*
- r_d *rounds are needed in order for all the bits of the state to depend on all the bits of the key.*

Then, we suggest using a number of rounds equal to $H_\eta = \lceil 2r_d + (1+\eta)r_e \rceil$*, where* η *is a* security factor *intended to take into account possible improvements of the relevant distinguishers.*

This method is heuristic as it is impossible to foresee how the best distinguishers will be improved, if at all. At the same time, we think it makes more sense than an approach based e.g. on simply doubling the number of rounds needed to prevent known distinguisher since it takes into account the actual structure of the attacks known. Our restriction to "relevant" distinguisher allows for example designers to discard related-key distinguisher if those are not relevant for their design. On the other hand, in our case, we consider related-tweak distinguishers to be relevant. In our definition of r_d, we assume that the diffusion is equally fast in the forward and in the backward direction.

A Lightweight Block Cipher. We can use our round function to build Crax-S-10, a lightweight block cipher operating on 64-bit using a 128-bit key intended for the most constrained micro-controllers. We claim that it provides 128 bits of security in the single-key setting. A reference implementation is provided in Appendix D.1. We used a security factor $\eta = 0.2$, so that the total number steps corresponds to $10 = \lceil 2 + 2 + r_e \times 1.2 \rceil$.

Our cipher uses a tweakless instance of the S step function described above. Since the step function has good diffusion and since we do not aim for related-key security, we use a very simple key schedule: the 64-bit round key k_i used at the beginning of step i is simply $k_i = K_{i \mod 2} \oplus i$, where the master key is (K_0, K_1). As there is no tweak, we do not need to worry about a bad interaction between tweak and key.

In order to prevent slide properties, we use the step counter in combination with a reduction of the number of round constants: instead of using all 8 of them, we only use 5. That way, in the first half of the cipher the steps involve c_i and $K_{i \mod 2}$ while in the second half they use c_i and $K_{(i+1) \mod 2}$. For other attacks, the security of Crax-S-10 follows directly from our analysis of Alzette.

Crax-S-10 is a very lightweight block cipher, arguably one of the the lightest ever reported in the literature when it comes to micro-controller implementations. The code size, RAM consumption, and execution time of Crax-S-10 on an 8-bit AVR and a 32-bit ARM Cortex-M3 micro-controller are summarized in Table 11, along with those of five other lightweight block ciphers, namely Speck-64/128 [7], Simon-64/128 [7], Rectangle [44], Sparx [19], and Hight [23]. We took the results for the five ciphers from the best implementations contained in the FELICS project [18], which is in the case of Speck the implementation "03" for ARM Cortex-M3 and the implementation "06" for the AVR architecture.[8] Since all these ciphers have a block size of 64 bits and a key size of 128 bits, they serve as good references for comparison.

The ARM implementation of Crax we benchmarked is exactly the optimized C code included in Appendix D.1. Encrypting a single 64-bit block on a Cortex-M3 takes 239 cycles (including function-call overhead), and the decryption has exactly the same execution time. The only serious competitor of Crax is Speck;

[8] The source code of these implementations and the complete benchmarking results are available on the CryptoLux wiki at http://www.cryptolux.org/index.php/FELICS_Block_Ciphers_Detailed_Results ("Scenario 0").

Table 11. A comparison of the implementation results of Crax-S-10 and five other lightweight ciphers with a block size of 64 bits and a key size of 128 bits. RAM and ROM consumption are measured in bytes and the time for processing a 64-bit block is given in clock cycles.

		Enc. 64-bit			Dec. 64-bit			Key schedule		
		ROM	RAM	Time	ROM	RAM	Time	ROM	RAM	Time
32-bit ARM (Cortex-M3)	Speck	340	132	184	448	132	254	48	132	514
	Crax-S	196	36	239	202	36	239	0	0	0
	Simon	100	200	557	120	200	596	244	200	864
	Rectangle	412	232	714	484	232	843	157	232	1106
	Sparx	644	224	932	748	224	1065	756	224	620
	Hight	442	160	2500	450	160	2644	352	160	740
8-bit AVR (ATmega128)	Speck	542	132	997	706	132	1139	178	132	1401
	Crax-S	584	20	1257	582	20	1249	0	0	0
	Simon	354	200	1973	364	200	1803	254	200	2911
	Rectangle	230	232	1823	230	232	1843	218	232	1832
	Sparx	712	224	1529	790	224	1676	642	224	844
	Hight	286	160	2770	288	160	2768	418	160	1191

all other ciphers given in Table 11 are much slower. Speck-64/128 encrypts and decrypts at a rate of 184 and 254 cycles per block, respectively. However, since Speck needs first to run its key schedule, Crax-S-10 encryption is faster than Speck for short messages of up to 9 blocks (i.e. 72 bytes). The Speck implementation occupies significantly more RAM than that of Crax (mostly because of the round keys) and has a much larger binary code size.

The results for the 8-bit AVR platform in Table 11 were all obtained with hand-written assembler implementations. The overall picture is similar to ARM, namely Speck and Crax clearly outperform the other four ciphers. When executed on an ATmega128 micro-controller, Crax-S-10 is slightly slower than Speck-64/128 when if we leave the key schedule aside, but is actually faster on short messages (up to 5 blocks). Similar to ARM, the round keys make Speck significantly more RAM-demanding than Crax. In terms of code size, the decryption Crax is smaller than that of Speck, while the encryption is slightly larger. However, when both functionalities are needed, Crax consumes less code space than Speck (including key schedule).

In summary, we can say that Crax is at least as light as Speck (lighter on ARM, comparable on AVR). Further, Crax shines for short messages, which are common in real-world applications like simple challenge-response protocols for the authentication of RFID tags and other IoT devices.

A Wide Tweakable Block Cipher. We can build an efficient software-oriented 256-bit tweakable block cipher with a 256-bit key and a 128-bit tweak using Trax-L-17 (pronounced "T-rax"). We claim related-tweak security as long as the total number of (x, T) queries to the encryption (or decryption) oracle for a given key k is at most equal to 2^{128}. We do not make any claim in the related-key setting. A reference implementation of the whole encryption is provided in Appendix D.2.

The motivation for this bound on the data complexity is simple: while an attacker may have tremendous computing power, it is impossible that they obtain this many plaintext/ciphertext pairs. Furthermore, the security of many modes of operations drops when the amount of queries reaches the birthday bound—2^{128} in our case. Combining the fact that the best distinguisher in the related-tweak setting cannot cover 9 steps with the same security factor as Crax-S-10 (namely $\eta = 0.2$), we use $\lceil 3 + 3 + 1.2 \times 9 \rceil = 17$ steps.

For the key schedule, we use a simple generalized Feistel structure to update the key state and thus derive $k_{i+1} = F_i(k_i)$, where k_0 is the 256-bit master key and where P_i is $\sigma \circ F_i$, with $\sigma(x_0, ..., x_7) = (x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_0)$ and

$$F_i(x_0, x_1, x_2, x_3, x_4, x_5, x_6, x_7) = \big(x_0 + x_1 + c_{2i+1},\ x_1,\ x_2 \oplus x_3 \oplus i,\ x_3,\\ x_4 + x_5 + c_{2i},\ x_5,\ x_6 \oplus x_7 \oplus (i \ll 16),\ x_7\big)\ ,$$

where the constant indices are taken modulo 8. This key schedule ensures that the key material undergoes some transformation so as to break potential patterns linking subkeys and tweak.

A tweakable block cipher lends itself well to a parallelizable mode of operation such as ΘCB [26], a variant of OCB which saves its complex overhead needed to turn a regular block cipher into a tweakable one. Since our block size is equal to 256 bits, attacks relying on collisions obtained via the birthday paradox are non-issue with Trax-L-17. Some modes such as the Synthetic Counter-in-Tweak [37] can retain a security level up to the birthday bound in case of nonce misuse. As suggested in [15], using a 256-bit block cipher can also help providing post-quantum security in cases where the attacker is given a lot of power (e.g. if the primitive runs on a quantum computer).[9]

In summary, Trax-L-17 can be used in SCT mode to provide 128 bits of security in case of nonce-misuse, and its large block size can frustrate some quantum attacks when used in the same mode as Saturnin: it can be used to offer a very robust authenticated encryption. On a Cortex-M3 microcontroller, the generation of subkeys, encryption, and decryption has an execution time of 925, 2435, and 2464 clock cycles, respectively. These results are based on a standard C implementation, whereby we determined the execution times with help of the cycle-accurate instruction set simulator of Keil Microvision v5.24.2 using a generic Cortex-M3 model as target device. For comparison, the currently fastest implementation of Saturnin has a (simulated) encryption time of 5494 cycles and a decryption time of 5489 cycles on a generic Cortex-M3 device.[10] Consequently, Trax-L-17 outperforms Saturnin by a factor of more than 2.2.

[9] We remark that in several modes of operations, like ΘCB, it is necessary to take care of domain separation. For instance, a few bits of the tweak can be reserved for this purpose. For example, the NIST lightweight AEAD candidate `SKINNY-AEAD` [10] simply dedicates one byte of the tweak for domain separation. Therefore, if a full 256-bit tweak needs to be exploited, a tweakable block cipher with a (slightly) larger tweak length of $256 + x$ would be beneficial.

[10] The source code of this implementation was developed by Rhys Weatherley and is available on GitHub at https://github.com/rweather/lightweight-crypto/blob/master/src/individual/Saturnin/saturnin.c.

The use of 32-bit operations implies that it is possible to vectorize the computation of several parallel Trax-L-17 instances on many platforms, meaning that its speed can be multiplied whenever e.g. AVX instructions are available.

6 Conclusion

Alzette is a component of a new kind, a wide S-box operating on 64 bits that can nevertheless be argued to provide strong security against many attacks. Because of its reliance on ARX operations with carefully chosen rotations, a constant-time implementation is both easy to write and very efficient on a wide class of processors and micro-controllers.

The NIST LWC submission Sparkle [8] provides the first application of the Alzette S-box, but we showed that Alzette can also be used to design software-efficient (tweakable) block ciphers on a variety of block lengths. A modified long-trail argument allows us to estimate the number of rounds needed to provide security with regard to (related-tweak) differential and linear attacks. We provided two concrete instances of this approach: the 64-bit block cipher Crax and the 256-bit tweakable block cipher Trax. Due to its very simple key schedule, Crax is competitive compared to the block cipher Speck: it consumes less RAM and is faster for short messages consisting of up to nine 64-bit blocks. On the other hand, the large block size of Trax can be used to obtain strong security guarantees in settings where the attacker is quite powerful (nonce-misuse, quantum computing) while its use of a tweak eases the use of parallelizable modes of operation that can better leverage vector instructions.

Acknowledgements. Part of the work of Christof Beierle was funded by *Deutsche Forschungsgemeinschaft (DFG)*, project number 411879806, and 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. The experiments presented in this paper were carried out using the HPC facilities of the University of Luxembourg [42] – see https://hpc.uni.lu.

References

1. AlTawy, R., Gong, G., He, M., Jha, A., Mandal, K., Nandi, M., Rohit, R.: SpoC: An authenticated cipher. NIST round 2 lightweight candidate, see also https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/spoc-spec-round2.pdf (2019)

2. AlTawy, R., Gong, G., He, M., Mandal, K., Rohit, R.: SPIX: An authenticated cipher. NIST round 2 lightweight candidate, see also https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/spix-spec-round2.pdf (2019)
3. AlTawy, R., Rohit, R., He, M., Mandal, K., Yang, G., Gong, G.: sLiSCP: Simeck-based permutations for lightweight sponge cryptographic primitives. In: Adams, C., Camenisch, J. (eds.) SAC 2017. LNCS, vol. 10719, pp. 129–150. Springer, Heidelberg (Aug 2017)
4. Altawy, R., Rohit, R., He, M., Mandal, K., Yang, G., Gong, G.: SLISCP-light: Towards hardware optimized sponge-specific cryptographic permutations. ACM Trans. Embed. Comput. Syst. 17(4), 81:1–81:26 (Aug 2018)
5. Barreto, P., Nikov, V., Nikova, S., Rijmen, V., Tischhauser, E.: Whirlwind: a new cryptographic hash function. Des. Codes Cryptogr. 56(2), 141–162 (2010)
6. Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., Wingers, L.: The SIMON and SPECK families of lightweight block ciphers. IACR Cryptology ePrint Archive 2013, 404 (2013), http://eprint.iacr.org/2013/404
7. Beaulieu, R., Shors, D., Smith, J., Treatman-Clark, S., Weeks, B., Wingers, L.: The SIMON and SPECK block ciphers on AVR 8-bit microcontrollers. Cryptology ePrint Archive, Report 2014/947 (2014), http://eprint.iacr.org/2014/947
8. Beierle, C., Biryukov, A., dos Santos, L.C., Großschädl, J., Perrin, L., Udovenko, A., Velichkov, V., Wang, Q.: Schwaemm and Esch: lightweight authenticated encryption and hashing using the Sparkle permutation family. NIST round 2 lightweight candidate, see also https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/sparkle-spec-round2.pdf (2019)
9. Beierle, C., Jean, J., Kölbl, S., Leander, G., Moradi, A., Peyrin, T., Sasaki, Y., Sasdrich, P., Sim, S.M.: The SKINNY family of block ciphers and its low-latency variant MANTIS. In: Robshaw, M., Katz, J. (eds.) CRYPTO 2016, Part II. LNCS, vol. 9815, pp. 123–153. Springer, Heidelberg (Aug 2016)
10. Beierle, C., Jean, J., Kölbl, S., Leander, G., Moradi, A., Peyrin, T., Sasaki, Y., Sasdrich, P., Sim, S.M.: SKINNY-AEAD and SKINNY-Hash. NIST round 2 lightweight candidate, see also https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/SKINNY-spec-round2.pdf (2019)
11. Bernstein, D.J., Kölbl, S., Lucks, S., Massolino, P.M.C., Mendel, F., Nawaz, K., Schneider, T., Schwabe, P., Standaert, F.X., Todo, Y., Viguier, B.: Gimli : A cross-platform permutation. In: Fischer, W., Homma, N. (eds.) CHES 2017. LNCS, vol. 10529, pp. 299–320. Springer, Heidelberg (Sep 2017)
12. Biham, E., Shamir, A.: Differential cryptanalysis of DES-like cryptosystems. In: Menezes, A.J., Vanstone, S.A. (eds.) CRYPTO'90. LNCS, vol. 537, pp. 2–21. Springer, Heidelberg (Aug 1991)
13. Biryukov, A., De Canniére, C., Braeken, A., Preneel, B.: A toolbox for cryptanalysis: Linear and affine equivalence algorithms. In: Biham, E. (ed.) EUROCRYPT 2003. LNCS, vol. 2656, pp. 33–50. Springer, Heidelberg (May 2003)
14. Biryukov, A., Velichkov, V., Corre, Y.L.: Automatic search for the best trails in ARX: Application to block cipher speck. In: Peyrin [36], pp. 289–310
15. Canteaut, A., Duval, S., Leurent, G., Naya-Plasencia, M., Perrin, L., Pornin, T., Schrottenloher, A.: Saturnin: a suite of lightweight symmetric algorithms for post-quantum security. NIST round 2 lightweight candidate, see also https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/saturnin-spec-round2.pdf (2019)

16. Cheon, J.H., Takagi, T. (eds.): ASIACRYPT 2016, Part I, LNCS, vol. 10031. Springer, Heidelberg (Dec 2016)
17. Dinu, D.: Efficient and Secure Implementations of Lightweight Symmetric Cryptographic Primitives. Ph.D. thesis, University of Luxembourg (2017), available online at https://orbilu.uni.lu/handle/10993/33803
18. Dinu, D., Corre, Y.L., Khovratovich, D., Perrin, L., Großschädl, J., Biryukov, A.: Triathlon of lightweight block ciphers for the internet of things. Journal of Cryptographic Engineering 9(3), 283–302 (Sep 2019)
19. Dinu, D., Perrin, L., Udovenko, A., Velichkov, V., Großschädl, J., Biryukov, A.: Design strategies for ARX with provable bounds: Sparx and LAX. In: Cheon and Takagi [16], pp. 484–513
20. Fu, K., Wang, M., Guo, Y., Sun, S., Hu, L.: MILP-based automatic search algorithms for differential and linear trails for speck. In: Peyrin [36], pp. 268–288
21. Gueron, S., Johnson, S., Walker, J.: SHA-512/256. Cryptology ePrint Archive, Report 2010/548 (2010), http://eprint.iacr.org/2010/548
22. Gurobi Optimization, L.: Gurobi optimizer reference manual (2018), http://www.gurobi.com
23. Hong, D., Sung, J., Hong, S., Lim, J., Lee, S., Koo, B., Lee, C., Chang, D., Lee, J., Jeong, K., Kim, H., Kim, J., Chee, S.: HIGHT: A new block cipher suitable for low-resource device. In: Goubin, L., Matsui, M. (eds.) CHES 2006. LNCS, vol. 4249, pp. 46–59. Springer (Oct 2006)
24. Knudsen, L.: Deal - a 128-bit block cipher. NIST AES Proposal (1998)
25. Kranz, T., Leander, G., Wiemer, F.: Linear cryptanalysis: Key schedules and tweakable block ciphers. IACR Trans. Symm. Cryptol. 2017(1), 474–505 (2017)
26. Krovetz, T., Rogaway, P.: The software performance of authenticated-encryption modes. In: Joux, A. (ed.) FSE 2011. LNCS, vol. 6733, pp. 306–327. Springer, Heidelberg (Feb 2011)
27. Leander, G., Abdelraheem, M.A., AlKhzaimi, H., Zenner, E.: A cryptanalysis of PRINTcipher: The invariant subspace attack. In: Rogaway, P. (ed.) CRYPTO 2011. LNCS, vol. 6841, pp. 206–221. Springer, Heidelberg (Aug 2011)
28. Liu, Y., Wang, Q., Rijmen, V.: Automatic search of linear trails in ARX with applications to SPECK and Chaskey. In: Manulis, M., Sadeghi, A.R., Schneider, S. (eds.) ACNS 16. LNCS, vol. 9696, pp. 485–499. Springer, Heidelberg (Jun 2016)
29. Liu, Z.: Automatic Tools for Differential and Linear Cryptanalysis of ARX Ciphers. Ph.D. thesis, University of Chinese Academy of Science (2017), in Chinese
30. Liu, Z., Li, Y., Jiao, L., Wang, M.: A new method for Searching Optimal Differential and Linear Trails in ARX Ciphers. Cryptology ePrint Archive, Report 2019/1438 (2019), https://eprint.iacr.org/2019/1438
31. Matsui, M.: Linear cryptanalysis method for DES cipher. In: Helleseth, T. (ed.) EUROCRYPT'93. LNCS, vol. 765, pp. 386–397. Springer, Heidelberg (May 1994)
32. Matsui, M.: On correlation between the order of S-boxes and the strength of DES. In: Santis, A.D. (ed.) EUROCRYPT'94. LNCS, vol. 950, pp. 366–375. Springer, Heidelberg (May 1995)
33. Niels, F., Lucks, S., Schneier, B., Whiting, D., Bellare, M., Kohno, T., Callas, J., Walker, J.: The Skein hash function family. Submission to the NIST SHA-3 competition (round 3) (2010)
34. Niemetz, A., Preiner, M., Biere, A.: Boolector 2.0 system description. Journal on Satisfiability, Boolean Modeling and Computation 9, 53–58 (2014 (published 2015)), https://github.com/boolector/boolector

35. Patarin, J., Goubin, L., Courtois, N.: Improved algorithms for isomorphisms of polynomials. In: Nyberg, K. (ed.) EUROCRYPT'98. LNCS, vol. 1403, pp. 184–200. Springer, Heidelberg (May / Jun 1998)
36. Peyrin, T. (ed.): FSE 2016, LNCS, vol. 9783. Springer, Heidelberg (Mar 2016)
37. Peyrin, T., Seurin, Y.: Counter-in-tweak: Authenticated encryption modes for tweakable block ciphers. In: Robshaw, M., Katz, J. (eds.) CRYPTO 2016, Part I. LNCS, vol. 9814, pp. 33–63. Springer, Heidelberg (Aug 2016)
38. Sun, L., Wang, W., Wang, M.: Automatic search of bit-based division property for ARX ciphers and word-based division property. In: Takagi, T., Peyrin, T. (eds.) ASIACRYPT 2017, Part I. LNCS, vol. 10624, pp. 128–157. Springer, Heidelberg (Dec 2017)
39. Todo, Y.: Structural evaluation by generalized integral property. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015, Part I. LNCS, vol. 9056, pp. 287–314. Springer, Heidelberg (Apr 2015)
40. Todo, Y., Leander, G., Sasaki, Y.: Nonlinear invariant attack - practical attack on full SCREAM, iSCREAM, and Midori64. In: Cheon, J.H., Takagi, T. (eds.) ASIACRYPT 2016, Part II. LNCS, vol. 10032, pp. 3–33. Springer, Heidelberg (Dec 2016)
41. Todo, Y., Morii, M.: Bit-based division property and application to simon family. In: Peyrin [36], pp. 357–377
42. Varrette, S., Bouvry, P., Cartiaux, H., Georgatos, F.: Management of an academic HPC cluster: The UL experience. In: Proc. of the 2014 Intl. Conf. on High Performance Computing & Simulation (HPCS 2014). pp. 959–967. IEEE, Bologna, Italy (July 2014)
43. Xiang, Z., Zhang, W., Bao, Z., Lin, D.: Applying MILP method to searching integral distinguishers based on division property for 6 lightweight block ciphers. In: Cheon and Takagi [16], pp. 648–678
44. Zhang, W., Bao, Z., Lin, D., Rijmen, V., Yang, B., Verbauwhede, I.: RECTANGLE: a bit-slice lightweight block cipher suitable for multiple platforms. Sci. China Inf. Sci. 58(12), 1–15 (2015)

A Linear Trails in Alzette

Table 12. The input and output masks α, β (in hex) of all linear trails over Alzette corresponding to maximum expected absolute linear trail correlation $c = 2^{-2}$ and $c = 2^{-5}$ for four and five rounds, respectively. The column $\max\{-\log_2(\tilde{c})\}$ represents the smallest observed correlations of the approximations taken over *all* (combinations of) Alzette instances that can occur without a step counter addition in SPARKLE. Similarly, the column $\min\{-\log_2(\tilde{c})\}$ represents the largest observed correlations of the approximations. In all of the experiments, the sample size was 2^{24}.

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

Table 13. The input and output masks α, β (in hex) of all linear trails over Alzette corresponding to maximum expected absolute linear trail correlation $c = 2^{-8}$ for six rounds. The column $\max\{-\log_2(\tilde{c})\}$ represents the smallest observed correlations of the approximations taken over *all* combinations of Alzette instances that can occur without a step counter addition in SPARKLE. Similarly, the column $\min\{-\log_2(\tilde{c})\}$ represents the largest observed correlations of the approximations. In all of the experiments, the sample size was 2^{28}.

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

B Algorithms

Algorithm 2 TRAX-S$_{n_s}$
In: $(x, y) \in \mathbb{F}_2^{32} \times \mathbb{F}_2^{32}, \quad K \in (\mathbb{F}_2^{64})^{n_s+1}, \quad T \in \mathbb{F}_2^{64}$
Out: $(x, y) \in \mathbb{F}_2^{32} \times \mathbb{F}_2^{32}$

$(c_0, c_1, c_2, c_3) \leftarrow$ (0xB7E15162,0xBF715880,0x38B4DA56,0x324E7738)
$(c_4, c_5, c_6, c_7) \leftarrow$ (0xBB1185EB,0x4F7C7B57,0xCFBFA1C8,0xC2B3293D)
for all $s \in [0, n_s - 1]$ **do**
  if $s = 1 \bmod 2$ **then**
    $(x, y) \leftarrow (x, y) \oplus T$
  end if
  $(x, y) \leftarrow (x, y) \oplus K_s$
  $(x, y) \leftarrow A_{c_s \bmod 8}(x, y)$
end for
$(x, y) \leftarrow (x, y) \oplus K_{n_s}$
return (x, y)

Algorithm 3 TRAX-M$_{n_s}$
In: $\big((x_0, y_0), (x_1, y_1)\big), x_i, y_i \in \mathbb{F}_2^{32}, \quad K \in (\mathbb{F}_2^{128})^{n_s+1}, \quad T \in \mathbb{F}_2^{64}$
Out: $\big((x_0, y_0), (x_1, y_1)\big), x_i, y_i \in \mathbb{F}_2^{32}$

$(c_0, c_1, c_2, c_3) \leftarrow$ (0xB7E15162,0xBF715880,0x38B4DA56,0x324E7738)
$(c_4, c_5, c_6, c_7) \leftarrow$ (0xBB1185EB,0x4F7C7B57,0xCFBFA1C8,0xC2B3293D)
for all $s \in [0, n_s - 1]$ **do**
  if $s = 1 \bmod 2$ **then**
    $(x_0, y_0) \leftarrow (x_0, y_0) \oplus T$
  end if
  $(x_0, y_0, x_1, y_1) \leftarrow (x_0, y_0, x_1, y_1) \oplus K_s$
  $(x_0, y_0) \leftarrow A_{c_{2s} \bmod 8}(x_0, y_0)$
  $(x_1, y_1) \leftarrow A_{c_{2s+1} \bmod 8}(x_1, y_1)$
  $(x_0, y_0, x_1, y_1) \leftarrow (x_0 \oplus x_1, y_0 \oplus y_1, x_0, y_0)$
end for
$(x_0, y_0, x_1, y_1) \leftarrow (x_0, y_0, x_1, y_1) \oplus K_{n_s}$
return $\big((x_0, y_0), (x_1, y_1)\big)$

Algorithm 4 TRAX-L$_{n_s}$

In: $\big((x_0, y_0), \ldots, (x_3, y_3)\big), x_i, y_i \in \mathbb{F}_2^{32}, \quad K \in (\mathbb{F}_2^{256})^{n_s+1}, \quad T \in \mathbb{F}_2^{128}$

Out: $\big((x_0, y_0), \ldots, (x_3, y_3)\big), x_i, y_i \in \mathbb{F}_2^{32}$

$(c_0, c_1, c_2, c_3) \leftarrow$ (0xB7E15162,0xBF715880,0x38B4DA56,0x324E7738)
$(c_4, c_5, c_6, c_7) \leftarrow$ (0xBB1185EB,0x4F7C7B57,0xCFBFA1C8,0xC2B3293D)
for all $s \in [0, n_s - 1]$ **do**
  if $s = 1 \bmod 2$ **then**
    $(x_0, y_0, x_1, y_1) \leftarrow (x_0, y_0, x_1, y_1) \oplus T$
  end if
  $((x_0, y_0), \ldots, (x_3, y_3)) \leftarrow ((x_0, y_0), \ldots, (x_3, y_3)) \oplus K_s$
  $(x_0, y_0) \leftarrow A_{c_{4s \bmod 8}}(x_0, y_0)$
  $(x_1, y_1) \leftarrow A_{c_{4s+1 \bmod 8}}(x_1, y_1)$
  $(x_2, y_2) \leftarrow A_{c_{4s+2 \bmod 8}}(x_2, y_2)$
  $(x_3, y_3) \leftarrow A_{c_{4s+3 \bmod 8}}(x_3, y_3)$
  $((x_0, y_0), \ldots, (x_3, y_3)) \leftarrow \mathcal{L}_4((x_0, y_0), \ldots, (x_3, y_3))$
end for
$((x_0, y_0), \ldots, (x_3, y_3)) \leftarrow ((x_0, y_0), \ldots, (x_3, y_3)) \oplus K_{n_s}$
return $\big((x_0, y_0), \ldots, (x_3, y_3)\big)$

C Assembly Implementation

The `ALZETTE` macro in ARM assembler is shown below. It uses only 12 instructions and all rotations are performed together with either an `add` or an `eor` (i.e. exclusive or) instruction. Consequently, no explicit rotation instructions have to be executed.

```
.macro ALZETTE xi:req, yi:req, ci:req
    add \xi, \xi, \yi, ror #31
    eor \yi, \yi, \xi, ror #24
    eor \xi, \xi, \ci
    add \xi, \xi, \yi, ror #17
    eor \yi, \yi, \xi, ror #17
    eor \xi, \xi, \ci
    add \xi, \xi, \yi
    eor \yi, \yi, \xi, ror #31
    eor \xi, \xi, \ci
    add \xi, \xi, \yi, ror #24
    eor \yi, \yi, \xi, ror #16
    eor \xi, \xi, \ci
.endm
```

D Implementation Details

In this section, we provide basic C implementations of Crax-S-10 and Trax-L-17. Both use the following macros for the computation of Alzette and its inverse.

```
#define ROT(x, n) (((x) >> (n)) | ((x) << (32-(n))))

#define ALZETTE(x, y, c)                    \
  (x) += ROT((y), 31), (y) ^= ROT((x), 24), \
  (x) ^= (c),                               \
  (x) += ROT((y), 17), (y) ^= ROT((x), 17), \
  (x) ^= (c),                               \
  (x) += (y),          (y) ^= ROT((x), 31), \
  (x) ^= (c),                               \
  (x) += ROT((y), 24), (y) ^= ROT((x), 16), \
  (x) ^= (c)

#define ALZETTE_INV(x, y, c)                \
  (x) ^= (c),                               \
  (y) ^= ROT((x), 16), (x) -= ROT((y), 24), \
  (x) ^= (c),                               \
  (y) ^= ROT((x), 31), (x) -= (y),          \
  (x) ^= (c),                               \
  (y) ^= ROT((x), 17), (x) -= ROT((y), 17), \
  (x) ^= (c),                               \
  (y) ^= ROT((x), 24), (x) -= ROT((y), 31)
```

D.1 Reference and Optimized Implementation of Crax-S

The listing below shows a reference C implementation of the encryption function of Crax-S-10 as described in Subsect. 5.2. Note that only the first 5 of the 8 round constants specified in Subsect. 2.4 are used by Crax-S.

```
#include "craxs10.h"

#define N_STEPS 10

static const uint32_t RCON[5] = {                          \
  0xB7E15162, 0xBF715880, 0x38B4DA56, 0x324E7738, 0xBB1185EB };

void craxs10_enc_ref(uint32_t *xword, uint32_t *yword, const uint32_t *key)
{
  int step;

  for (step = 0; step < NSTEPS; step++) {
    xword[0] ^= step;
    if ((step % 2) == 0) {
      xword[0] ^= key[0];
      yword[0] ^= key[1];
    } else {
      xword[0] ^= key[2];
      yword[0] ^= key[3];
    }
    ALZETTE(xword[0], yword[0], RCON[step%5]);
  }
  xword[0] ^= key[0];
  yword[0] ^= key[1];
}
```

The decryption performs the same operations as the encryption, but in reverse order, and uses the inverse of Alzette. A reference C implementation of the decryption is given below.

```c
void craxs10_dec_ref(uint32_t *xword, uint32_t *yword, const uint32_t *key)
{
  int step;

  xword[0] ^= key[0];
  yword[0] ^= key[1];
  for (step = NSTEPS-1; step >= 0; step--) {
    ALZETTE_INV(xword[0], yword[0], RCON[step%5]);
    if ((step % 2) == 0) {
      xword[0] ^= key[0];
      yword[0] ^= key[1];
    } else {
      xword[0] ^= key[2];
      yword[0] ^= key[3];
    }
    xword[0] ^= step;
  }
}
```

The optimized C implementation of the encryption function differs from its reference counterpart in three main aspects to improve the execution time. First, the optimized C code advises the compiler to keep the state words x, y and the words of the key in registers. Second, the optimized code partially unrolls the loop to eliminate the `if-then-else` clause of the reference implementation and reduce the loop overhead. Third, it duplicates the five state words to avoid the modulo-5 reduction in the calculation of the index through which the round constant is loaded. These optimizations reduce the execution time significantly at the expense of a slight increase of code size. The following listing contains the optimized C code of the encryption function.

```c
static const uint32_t RCON[10] = {                              \
  0xB7E15162, 0xBF715880, 0x38B4DA56, 0x324E7738, 0xBB1185EB, \
  0xB7E15162, 0xBF715880, 0x38B4DA56, 0x324E7738, 0xBB1185EB };

void craxs10_enc_opt(uint32_t *state, const uint32_t *key)
{
  register uint32_t xw = state[0], yw = state[1];
  register uint32_t k0 = key[0], k1 = key[1], k2 = key[2], k3 = key[3];
  int step = 0;

  while (step < NSTEPS) {
    xw ^= (k0 ^ step);
    yw ^= k1;
    ALZETTE(xw, yw, RCON[step]);
    step++;
    xw ^= (k2 ^ step);
    yw ^= k3;
    ALZETTE(xw, yw, RCON[step]);
    step++;
  }
  xw ^= k0;
  yw ^= k1;
  state[0] = xw;
  state[1] = yw;
}
```

The decryption can be optimized in a very similar way as described above for the encryption. The resulting C code is shown below.

```c
void craxs10_dec_opt(uint32_t *state, const uint32_t *key)
{
  register uint32_t xw = state[0], yw = state[1];
  register uint32_t k0 = key[0], k1 = key[1], k2 = key[2], k3 = key[3];
  int step = NSTEPS-1;

  xw ^= k0;
  yw ^= k1;
  while (step > 0) {
    ALZETTE_INV(xw, yw, RCON[step]);
    xw ^= (k2 ^ step);
    yw ^= k3;
    step--;
    ALZETTE_INV(xw, yw, RCON[step]);
    xw ^= (k0 ^ step);
    yw ^= k1;
    step--;
  }
  state[0] = xw;
  state[1] = yw;
}
```

D.2 Reference Implementation of Trax-L

The following listing shows the reference implementation of the Trax-L-17 functions for the generation of subkeys, encryption, and decryption, respectively. The macros for Alzette and its inverse are the same as for Crax-S-10.

```c
#include "traxl17.h"

#define NSTEPS 17
#define ELL(x) (ROT(((x) ^ ((x) << 16)), 16))

static const uint32_t RCON[8] = {                \
  0xB7E15162, 0xBF715880, 0x38B4DA56, 0x324E7738, \
  0xBB1185EB, 0x4F7C7B57, 0xCFBFA1C8, 0xC2B3293D };

void traxl17_genkeys_ref(uint32_t *subkeys, const uint32_t *key)
{
  uint32_t key_[8], tmp;
  int b, s;  // branch and step counter

  memcpy(key_, key, 32);
  for (s = 0; s < NSTEPS+1; s++) {
    // assign 8 sub-keys
    for (b = 0; b < 8; b++)
      subkeys[8*s+b] = key_[b];
    // update master-key
    key_[0] += key_[1] + RCON[(2*s)%8];
    key_[2] ^= key_[3] ^ s;
    key_[4] += key_[5] + RCON[(2*s+1)%8];
    key_[6] ^= key_[7] ^ ((uint32_t) (s << 16));
    // rotate master-key
    tmp = key_[0];
    for (b = 1; b < 8; b++)
      key_[b-1] = key_[b];
    key_[7] = tmp;
  }
```

```
}

void traxl17_enc_ref(uint32_t *x, uint32_t *y, const uint32_t *subkeys,
                     const uint32_t *tweak)
{
  uint32_t tmpx, tmpy;
  int b, s;  // branch and step counter

  for (s = 0; s < NSTEPS; s++) {
    // add tweak to state if step-counter is odd
    if ((s % 2) == 1) {
      x[0] ^= tweak[0]; y[0] ^= tweak[1];
      x[1] ^= tweak[2]; y[1] ^= tweak[3];
    }
    // add subkeys to state and execute ALZETTEs
    for (b = 0; b < 4; b++) {
      x[b] ^= subkeys[8*s+2*b];
      y[b] ^= subkeys[8*s+2*b+1];
      ALZETTE(x[b], y[b], RCON[(4*s+b)%8]);
    }
    // linear layer (see Sparkle256 permutation)
    tmpx = ELL(x[2] ^ x[3]); y[0] ^= tmpx; y[1] ^= tmpx;
    tmpy = ELL(y[2] ^ y[3]); x[0] ^= tmpy; x[1] ^= tmpy;
    tmpx = x[0]; x[0] = x[3]; x[3] = x[1]; x[1] = x[2]; x[2] = tmpx;
    tmpy = y[0]; y[0] = y[3]; y[3] = y[1]; y[1] = y[2]; y[2] = tmpy;
  }
  // add subkeys to state for final key addition
  for (b = 0; b < 4; b++) {
    x[b] ^= subkeys[8*NSTEPS+2*b];
    y[b] ^= subkeys[8*NSTEPS+2*b+1];
  }
}

void traxl17_dec_ref(uint32_t *x, uint32_t *y, const uint32_t *subkeys,
                     const uint32_t *tweak)
{
  uint32_t tmpx, tmpy;
  int b, s;  // branch and step counter

  // add subkeys to state for initial key addition
  for (b = 0; b < 4; b++) {
    y[b] ^= subkeys[8*NSTEPS+2*b+1];
    x[b] ^= subkeys[8*NSTEPS+2*b];
  }
  for (s = NSTEPS-1; s >= 0; s--) {
    // inverse linear layer (see Sparkle256 permutation)
    tmpy = y[0]; y[0] = y[2]; y[2] = y[1]; y[1] = y[3]; y[3] = tmpy;
    tmpx = x[0]; x[0] = x[2]; x[2] = x[1]; x[1] = x[3]; x[3] = tmpx;
    tmpy = ELL(y[2] ^ y[3]); x[0] ^= tmpy; x[1] ^= tmpy;
    tmpx = ELL(x[2] ^ x[3]); y[0] ^= tmpx; y[1] ^= tmpx;
    // add subkeys to state and execute inverse ALZETTEs
    for (b = 0; b < 4; b++) {
      ALZETTE_INV(x[b], y[b], RCON[(4*s+b)%8]);
      y[b] ^= subkeys[8*s+2*b+1];
      x[b] ^= subkeys[8*s+2*b];
    }
    // add tweak to state if step-counter is odd
    if ((s % 2) == 1) {
      y[1] ^= tweak[3]; x[1] ^= tweak[2];
      y[0] ^= tweak[1]; x[0] ^= tweak[0];

    }
  }
}
```

E Division Property Propagation in Trax Instances

We analyzed Trax-S, Trax-M, Trax-S structures using the bit-based division property. We used both the Gurobi MILP-solver and the Boolector SMT-solver.

Finding best integral distinguishers with limited data is a difficult task due to combinatorial complexity, namely, the amount of possible plaintext bit activation patterns is very large. The situation worsens even more due to the tweak, which we also include in our analysis. Therefore, we set to use the maximum possible amount of data and activate all bits except one; this will also result in strongest integral distinguishers.

Note that in a tweakable cipher the sum of all ciphertexts over all plaintexts with a fixed tweak is zero, due to the fixed-tweak tweakable cipher being a permutation. This does not contradict the security model of an ideal tweakable block cipher. However, it means that, in order to ignore this trivial distinguisher, we need to have at least one inactive bit in the plaintext. Then, even by activating in addition the whole tweak no trivial distinguisher is obtained. We conclude that such activation pattern - all but one plaintext bits and all tweak bits - leads to the strongest division property-based related-tweak integral distinguishers. Finally, we remark that the amount of data largely exceeds the amounts we allow for cryptanalysis (see Section 5.1). As a result, we obtain a rather loose upper bound on the number of rounds needed to prevent division property-based related-tweak integral attacks.

We chose to deactivate the bit which leads to best integral attacks on the keyless Alzette instance (see Section 3.3): the bit with index 44 for the forward direction and the bit with index 27 for the inverse direction (note that such integral distinguishers can not be concatenated in order to mount the permutation distinguisher, making the bounds even looser). We evaluated all possible choices of the input ARX-box to put the inactive bit to. For the inverse instances, we added the tweak after *even*-indexed inverse steps, so that if the normal instance is composed with the inverse instance, the period of two steps between tweak additions is maintained. Our results are summarized in Table 14.

It is worth noting that adding tweak to the attack surface results in about one 2 step improvement for 64-bit primitives: see Trax-S compared to pure Alzette (see Section 3.3). Also it is clear that the best distinguishers on Trax-M and Trax-L are structure-based (as opposed to bit-based), since whole 64-bit blocks are balanced. This supports our claim that Alzette itself has strong resistance against integral attacks.

Table 14. Best integral distinguishers for TRAX structures. `?` denotes possibly non-balanced 32-bit word, `B` means all 32 bits balanced, `b` means some of 32 bits are balanced. A step of TRAX-M/TRAX-L includes a layer of Alzette (4 rounds) and the linear layer.

instance	inactive bit	number of rounds	balanced pattern
Alzette	44	1 steps + 2 rounds	`bb`
Alzette^{-1}	27	1 steps + 1 rounds	`bb`
TRAX-S	44	3 steps + 2 rounds	`bb`
TRAX-S^{-1}	27	3 steps + 3 rounds	`?b`
TRAX-M	44	5 steps + 4 rounds	`??BB`
	108	3 steps + 4 rounds	`??BB`
TRAX-M^{-1}	27	4 steps + 0 rounds	`BB??`
	91	4 steps + 0 rounds	`BB??`
TRAX-L	44	6 steps + 4 rounds	`????BBBB`
	108	6 steps + 4 rounds	`????BBBB`
	172	5 steps + 4 rounds	`????BBBB`
	236	5 steps + 4 rounds	`????BBBB`
TRAX-L^{-1}	27	6 steps + 0 rounds	`BBBB????`
	91	6 steps + 0 rounds	`BBBB????`
	155	6 steps + 0 rounds	`BBBB????`
	219	6 steps + 0 rounds	`BBBB????`