Alzette: A 64-bit ARX-box

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

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

sparklegrupp@googlegroups.com

Abstract. 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.
In this paper, 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, while two iterations are at least as secure as the AES super S-box.
Since 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.

1 Introduction

Symmetric primitives need to be non-linear. It is common to rely on so-called *S-boxes* to obtain this property. 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 [1] used a 16-bit S-box based on the multiplicative inverse in the finite field $\mathbb{F}_{2^{16}}$. In this case, the intention was not for implementers 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 [8] and in the NIST lightweight candidate Saturnin [6]. 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.

It 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[5] to prove that the cipher meets some security criteria, it is not the case for the *ARX-box* of Sparx. Indeed, in order to implement the security argument designed by its authors (a *long trail 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 [5] 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 analyse 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.

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)

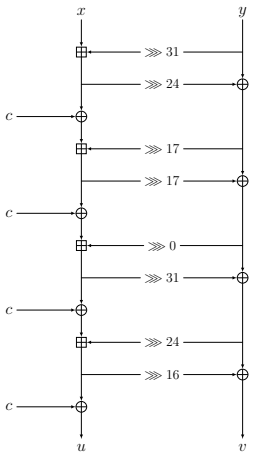

Fig. 1: The Alzette instance A_c.

Despite their superficial resemblance, Alzette has many differences with the Sparx ARX-box:

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

– it relies on 32- rather than 16-bit operations, meaning that it is suitable for a larger number of architectures;
– it makes better use of barrel shift registers (when they are available) and has more efficient rotation constants (for platforms on which they have different costs);
– it uses different rotations in each of its 4 rounds;
– 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;
– Alzette is not key-dependent and we studied the influence the constant it uses has on its properties.

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 this analysis for the round constants employed in the permutation Sparkle, submitted to the NIST lightweight cryptography standardization process [2]. 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 [21]. The source code for our experimental analysis can be found at https://github.com/cryptolu/sparkle.

Outline. The design process that we used to construct Alzette is explained in Section 2. In particular, we show that it offers resilience against many different attacks. This analysis is confirmed experimentally in Section 3. We also discuss the efficiency of Alzette in Section 4.

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 operation. Further, by $x \ggg r$, we denote the cyclic rotation of the 32-bit word x to the right by the offset r.

2 The Design of Alzette

In this section, we present both the design process and the main properties of Alzette. These are verified experimentally later in Section 3, and summarized in Section 3.5.

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. [7, Appendix A, B, C] for details).

While 16-bit words are sub-optimal 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.

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.[6] 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 [10] 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 a 32-bit word size

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

allows peak performance on 64-bit processors was already used for instance by the designers of Gimli [3].

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.

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 imposed 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 of the form depicted in Figure 2, where each word is of size 32 bits and which iterates t rounds. The i-th round is defined by the rotation amounts $(r_i, s_i) \in \mathbb{Z}_{32} \times \mathbb{Z}_{32}$ and the round constant $(\gamma_i^L, \gamma_i^R) \in \mathbb{F}_2^{32} \times \mathbb{F}_2^{32}$.

In our final design, we decided to use $t = 4$ rounds. The reason is that, 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

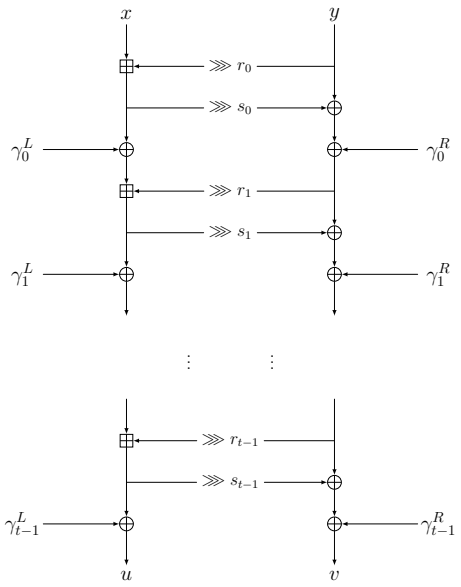

Fig. 2: The general structure of the ARX-box.

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

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

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

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

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

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

We tested *all* rotation sets with a cost below 12 for the above conditions. None of those fulfilled the above criteria. For a cost below 15, we found the ARX-box with the rotations as presented in Table 2. The first two lines correspond to the final choice of Alzette.

Table 2: Differential and linear bounds for our choice of rotation parameters with all four offsets. For each offset, the first line shows the differential bound and the second shows the linear one. The value set in parenthesis corresponds to the maximum absolute correlation of the linear hull taking clustering into account (see Section 3.2). The differential bounds [5] and linear bounds [9,12] 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	≥ 32	–	–	–
	0	0	1	3	6	9	13	17	19	21	24	27

2.4 On the Differential Properties

We used Algorithm 1 of [5] and adapted it to our round structure to compute the bounds on the maximum expected differential trail probabilities of the ARX-boxes we considered. This algorithm is basically a refined variant of Matsui's well-known branch and bound algorithm [13]. 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 [5] 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 up to 6 round and experimentally checked the actual probabilities of the differentials (for the constants used in SPARKLE), see Section 3.1.

Note that for 7 and 8 rounds, we could not get tight bounds 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 greater than 2^{-24} and 2^{-32}, respectively. We evaluated similar bounds for up to 12 rounds.

2.5 On the Linear Properties

We used the Mixed-Integer Linear Programming approach described in [9] 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 constants used in Sparkle), see Section 3.2.

2.6 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. Furthermore, and perhaps even more importantly, they should 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 three subsections is dependent on the actual choice of round constants c for Sparkle. Those constants are provided in Table 3.

2.7 Invariant Subspaces

Invariant subspace attacks were considered in [11]. For the round constants used in Sparkle, using a similar "to and fro" method from [14,4], 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 denote the first two rounds of A_{c_0}. Then for all $l, r, l', r' \in \mathbb{F}_2^{32}$ such that $A(l, r) = (l', r')$, it holds that

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

where "+" denotes a XOR. This equation defines a subspace trail of constant dimension 59.

2.8 Nonlinear Invariants

Nonlinear invariant attacks were considered recently in [19] 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}.

2.9 Linearization

In recent attack against KECCAK instances [16,17], the *S-box linearization* technique is used. The idea is to find a subset of inputs (often an affine subspace), such that the S-box acts linearly on this set. We attempted to linearize the ARX-boxes by finding all inputs for which all four modular additions inflict no carry bits and thus are equivalent to XOR. For the addition of two random independent 32-bit words, the probability of having all carry bits equal to zero is equal to $(3/4)^{31}$. Indeed, for each bit position, if no carry comes in, then the outgoing carry will occur only if both input bits are equal to 1. Furthermore, the carry bit from the most significant bits is ignored. Assuming independence of the additions in the ARX-box, $2^{64}/(3/4)^{124} \approx 2^{12.5}$ inputs are expected to satisfy the linearization.

In order to find all inputs, we have to solve a system of quadratic equations. Indeed, for the first round, the condition is $(x \& (y \ggg 31)) \ll 1 = 0$ (left shift by one omits the most significant bit), which provides 31 quadratic bit equations. Since this condition ensures that the output of the first round is linear, we get similar quadratic equations for the second round, except that x and y are replaced with corresponding linear functions. In total we obtain 124 quadratic equations of the form $l(x, y) \cdot r(x, y) = 0$, where $l, r \colon \mathbb{F}_2^{64} \to \mathbb{F}_2$ are affine. We solved this system by a guess-and-determine method with a few optimizations, for all round constants used in SPARKLE. The results are given in Table 3.

Table 3: The number of inputs for Alzette inflicting no carries in all four rounds, for different round constants. The constants c_0 up to c_7 are the round constants used in SPARKLE.

constant	hexadecimal	number of inputs
c_0	b7e15162	13
c_1	bf715880	11
c_2	38b4da56	18
c_3	324e7738	3
c_4	bb1185eb	10
c_5	4f7c7b57	340
c_6	cfbfa1c8	105
c_7	c2b3293d	76
0	00000000	8

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

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

3 Experimental Verifications

The following experimental verifications are done for the round constants used in SPARKLE, except those described in Section 3.4 which are independent of the choice of the constants. However, those methods can easily be applied to arbitrary choices of constants.

3.1 Experiments on the Fixed-Key Differential Probabilities

As in virtually all block cipher designs, the security arguments against differential attacks are only average results when *averaging over all keys of the primitive*. When leveraging such arguments for a cryptographic permutation, i.e., a block cipher with a fixed key, it might be possible in theory that the actual fixed-key

maximum differential probability is higher than the expected maximum differential probability. In particular, the variance of the distribution of the maximum fixed-key differential probabilities might be high.

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

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

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

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

For 5 rounds, i.e., one 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}.

3.2 Experiments on the Fixed-Key Linear Correlations

Similarly as for the case of differentials, for all of the 8 Alzette instances used in Sparkle, we conducted experiments in order to see whether the maximum expected absolute linear trail correlations derived by MILP and presented in Table 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 8 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

[7] 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 4: The input and output differences α, β (in hex) of all differential trails over Alzette corresponding to maximum expected differential trail probability $p = 2^{-6}$ and $p = 2^{-10}$ for four and five rounds, respectively.

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

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 8. 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.[8]

This tendency also holds for the correlations over six rounds. There are 48 trails with a maximum expected absolute linear trail correlation of 2^{-8}. The results of our experiments for 2^{28} random samples are shown in Table 9 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 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

[8] 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}.

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 correlations since our estimate is only $2^{1.21}$ times lower than the actual probability.

3.3 Experimental Algebraic Degree Lower Bound

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

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

3.4 Division Property of the ARX-box Structure

We performed a MILP-aided bit-based division property analysis [18,20] on the structure of Alzette. The MILP encoding is rather straightforward. For the modular addition operation we used the following method.

Addition modulo 2^{32}. We encode it by encoding the carry propagation. For any $a, b, c \in \mathbb{F}_2$, let $c' = \mathrm{Maj}(a, b, c) \in \mathbb{F}_2$ and $y = a \oplus b \in \mathbb{F}_2$. Then, all possible such 5-tuples $(a, b, c, c', y) \in \mathbb{F}_2^5$ can be characterized by the two following integer inequalities:

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

For any bit position, summing the input bits a, b with the input carry c results in the output bit y and the new carry c'. In our experiments, these two inequalities applied for each bit position generated precisely the correct division property table of addition modulo 2^n for n up to 7. There were some redundant transitions though, which do not affect the result.

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 5. Even 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 [20]. 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.

Table 5: The upper bounds on the minimum and maximum degree of the coordinates of Alzette and its inverse.

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

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

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

input active bits:

`11111111111111111111111111111111,11111111111101111111111111111111`,

balanced bits after 6-round ARX-box:

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

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

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

input active bits:

`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.

3.5 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 6). 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, the differential trail bound is 2^{-30} and the linear one is 2^{-15}: as the branching number of the MixColumn operation is 5, we raise the quantities of the S-box to the power 5.

Table 6: 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

Although Alzette was designed to be efficient in software, we briefly describe the implementation characteristics for both software and hardware implementations in the following.

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 $z \leftarrow x <op> (y \lll n)$ can be executed with a single instruction in a single clock cycle, irrespective of the rotation distance.

Alzette itself operates over two 32-bit words of data, with an extra 32-bit constant value. This allows the full computation to happen in-register in AVR, MSP and ARM architectures, whereby the latter is able to hold at least 4 Alzette instances entirely in registers. This in turn reduces load-store overheads and contributes to the performance of 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 7: Execution time (in clock cycles) and codes size (in bytes) of Alzette on an 8-bit AVR ATmega128 and a 32-bit ARM Cortex-M3 microcontroller.

Platform	Execution time	Code size
8-bit AVR	122	176
32-bit ARM	12	24

Table 7 summarizes the execution time and code size of Alzette on an 8-bit AVR and a 32-bit ARM Cortex-M3 microcontroller. The assembler implementation of Alzette for the latter architecture consists of 12 instructions (see Appendix B), which take 12 clock cycles to execute. The Cortex-M3 supports Thumb2, which means the used instructions are only 16 bits long. Consequently, the code size of Alzette is 24 bytes. 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 an ARM Cortex-M3 processor is big enough to accommodate, for example, a full 384-bit state.

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, which results in a code size of 156 bytes. However, in contrast to ARM, we can not assume that the whole state fits into the register file of an

AVR microcontroller (for example, a 384-bit state is too big for the register space of AVR), 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. Hence, loading a two 32-bit branches costs 16 cycles altogether, and writing them back to RAM costs another 16 cycles. Loading a 32-bit round constant from Flash requires 12 cycles. Consequently, when taking all loads/stores into account, the execution time increases from 78 to 122 cycles and the code size from 156 to 176 bytes.

4.2 Hardware Implementations

A 32-bit ALU is needed 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 Conclusion

Alzette is a component of a new kind, a wide S-box operating on 64 bits that can nevertheless be proven 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 microcontrollers.

The NIST LWC submission Sparkle [2] provides the first application of the Alzette S-box. Other ciphers could easily be built. For example, we could directly construct an Even-Mansour like 64-bit block cipher where round keys would be added in the full state in-between calls to Alzette.

Acknowledgements. The work of Christof Beierle was funded by the SnT CryptoLux RG budget. Luan Cardoso dos Santos is supported by the Luxembourg National Research Fund through grant PRIDE15/10621687/SPsquared. The work of Aleksei Udovenko was 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.

References

1. Barreto, P., Nikov, V., Nikova, S., Rijmen, V., Tischhauser, E.: Whirlwind: a new cryptographic hash function. Designs, Codes and Cryptography 56(2), 141–162 (2010), http://dx.doi.org/10.1007/s10623-010-9391-y
2. 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)
3. 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)
4. 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)
5. Biryukov, A., Velichkov, V., Corre, Y.L.: Automatic search for the best trails in ARX: Application to block cipher speck. In: Peyrin [15], pp. 289–310
6. 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)
7. 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
8. Dinu, D., Perrin, L., Udovenko, A., Velichkov, V., Großschädl, J., Biryukov, A.: Design strategies for ARX with provable bounds: Sparx and LAX. In: Cheon, J.H., Takagi, T. (eds.) ASIACRYPT 2016, Part I. LNCS, vol. 10031, pp. 484–513. Springer, Heidelberg (Dec 2016)
9. Fu, K., Wang, M., Guo, Y., Sun, S., Hu, L.: MILP-based automatic search algorithms for differential and linear trails for speck. In: Peyrin [15], pp. 268–288
10. Gueron, S., Johnson, S., Walker, J.: SHA-512/256. Cryptology ePrint Archive, Report 2010/548 (2010), http://eprint.iacr.org/2010/548
11. 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)

12. 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)
13. 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)
14. 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)
15. Peyrin, T. (ed.): FSE 2016, LNCS, vol. 9783. Springer, Heidelberg (Mar 2016)
16. Qiao, K., Song, L., Liu, M., Guo, J.: New collision attacks on round-reduced keccak. In: Coron, J., Nielsen, J.B. (eds.) EUROCRYPT 2017, Part III. LNCS, vol. 10212, pp. 216–243. Springer, Heidelberg (Apr / May 2017)
17. Song, L., Liao, G., Guo, J.: Non-full sbox linearization: Applications to collision attacks on round-reduced keccak. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017, Part II. LNCS, vol. 10402, pp. 428–451. Springer, Heidelberg (Aug 2017)
18. 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)
19. 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)
20. Todo, Y., Morii, M.: Bit-based division property and application to simon family. In: Peyrin [15], pp. 357–377
21. 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)

A Linear Trails in Alzette

Table 8: 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 9: 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 Assembly Implementation

The `ARX_BOX` 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 ARX_BOX 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
```