Reverse-Engineering the S-Box of Streebog, Kuznyechik and STRIBOBr1 (Full Version)*

Alex Biryukov[1], Léo Perrin[2], and Aleksei Udovenko[3]

[1] alex.biryukov@uni.lu, University of Luxembourg
[2] leo.perrin@uni.lu, SnT,University of Luxembourg
[3] aleksei.udovenko@uni.lu, SnT,University of Luxembourg

Abstract. The Russian Federation's standardization agency has recently published a hash function called Streebog and a 128-bit block cipher called Kuznyechik. Both of these algorithms use the same 8-bit S-Box but its design rationale was never made public.

In this paper, we reverse-engineer this S-Box and reveal its hidden structure. It is based on a sort of 2-round Feistel Network where exclusive-or is replaced by a finite field multiplication. This structure is hidden by two different linear layers applied before and after. In total, five different 4-bit S-Boxes, a multiplexer, two 8-bit linear permutations and two finite field multiplications in a field of size 2^4 are needed to compute the S-Box.

The knowledge of this decomposition allows a much more efficient hardware implementation by dividing the area and the delay by 2.5 and 8 respectively. However, the small 4-bit S-Boxes do not have very good cryptographic properties. In fact, one of them has a probability 1 differential.

We then generalize the method we used to partially recover the linear layers used to whiten the core of this S-Box and illustrate it with a generic decomposition attack against 4-round Feistel Networks whitened with unknown linear layers. Our attack exploits a particular pattern arising in the Linear Approximations Table of such functions.

Keywords: Reverse-Engineering, S-Box, Streebog, Kuznyechik, STRIBOBr1, White-Box, Linear Approximation Table, Feistel Network

1 Introduction

S-Boxes are key components of many symmetric cryptographic primitives including block ciphers and hash functions. Their use allows elegant security arguments

* Full version of the paper published in the proceedings of EuroCrypt 2016. The work of Léo Perrin is supported by the CORE ACRYPT project (ID C12-15-4009992) funded by the *Fonds National de la Recherche* (Luxembourg). The work of Aleksei Udovenko is supported by the *Fonds National de la Recherche*, Luxembourg (project reference 9037104)

based on the so-called wide-trail strategy [1] to justify that the primitive is secure against some of the best known attacks, e.g. differential [2] and linear [3,4] cryptanalysis.

Given the importance of their role, S-Boxes are carefully chosen and the criteria or algorithm used to build them are explained and justified by the designers of new algorithms. For example, since the seminal work of Nyberg on this topic [5], the inverse function in the finite field of size 2^n is often used (see the Advanced Encryption Standard [1], TWINE [6]...).

However, some algorithms are designed secretly and, thus, do not justify their design choices. Notable such instances are the primitives designed by the American National Security Agency (NSA) or standardized by the Russian Federal Agency on Technical Regulation and Metrology (FATRM). While the NSA eventually released some information about the design of the S-Boxes of the Data Encryption Standard [7,8], the criteria they used to pick the S-Box of Skipjack [9] remain mostly unknown despite some recent advances on the topic [10]. Similarly, recent algorithms standardized by FATRM share the same function π, an unexplained 8-bit S-Box. These algorithms are:

Streebog (officially called "GOST R 34.11-2012", sometimes spelled Stribog) is the new standard hash function for the Russian Federation [11]. Several cryptanalyses against this algorithm have been published. A second preimage attack requiring 2^{266} calls to the compression function instead of the expected 2^{512} has been found by Guo et al. [12]. Another attack [13] targets a modified version of the algorithm where only the round constants are modified: for some new round constants, it is actually possible to find collisions for the hash function. To show that the constants were not chosen with malicious intentions, the designers published a note [14] describing how they were derived from a modified version of the hash function. While puzzling at a first glance, the seeds actually correspond to Russian names written backward (see Appendix A).

Kuznyechik (officially called "GOST R 34.12-2015", sometimes spelled Kuznechik) is the new standard block cipher for the Russian Federation. It was first mentioned in [15] and is now available at [16]. It is a 128-bit block cipher with a 256-bit key consisting of 9 rounds of a Substitution-Permutation Network where the linear layer is a matrix multiplication in $(\mathbb{F}_{2^8})^{16}$ and the S-Box layer consists in the parallel application of an 8-bit S-Box. The best attack so far is a Meet-in-the-Middle attack covering 5 rounds [17]. It should not be mistaken with GOST 28147-89 [18], a 64-bit block cipher standardized in 1989 and which is sometimes referred to as "the GOST cipher" in the literature.

STRIBOB [19] is a CAESAR candidate which made it to the second round of the competition. The designer of this algorithm is not related to the Russian agencies. Still, the submission for the first round (STRIBOBr1) is based on Streebog.[4]

[4] The version submitted to the next round, referred to as "STRIBOBr2" and "WHIRLBOB", uses the S-Box of the Whirlpool hash function [20] whose design crite-

The Russian agency acting among other things as a counterpart of the American NSA is the Federal Security Service (FSB). It was officially involved in the design of Streebog. Interestingly, in a presentation given at RusCrypto'13 [22] by Shishkin on behalf of the FSB, some information about the design process of the S-Box is given: it is supposed not to have an analytic structure — even if that means not having optimal cryptographic properties unlike e.g. the S-Box of the AES [1] — and to minimize the number of operations necessary to compute it so as to optimize hardware and vectorized software implementations. However, the designers did not publish any more details about the rationale behind their choice for π and, as a consequence, very little is known about it apart from its look-up table which we give in Appendix B. In [23], Saarinen et al. summarize a discussion they had with some of the designers of the GOST algorithms at a conference in Moscow:

> We had brief informal discussions with some members of the Streebog and Kuznyechik design team at the CTCrypt'14 workshop (05-06 June 2014, Moscow RU). Their recollection was that the aim was to choose a "randomized" S-Box that meets the basic differential, linear, and algebraic requirements. Randomization was simply iterated until a "good enough" permutation was found. This was seen as an effective countermeasure against yet-unknown attacks [as well as algebraic attacks].

Since we know little to nothing about the design of this S-Box, it is natural to try and gather as much information as we can from its look-up table. In fact, the reverse-engineering of algorithms with unknown design criteria is not a new research area. We can mention for example the work of the community on the American National Security Agency's block cipher Skipjack [9] both before and after its release [24,25,26]. More recently, Biryukov et al. proved that its S-Box was not selected from a collection of random S-Boxes and was actually the product of an algorithm that optimized its linear properties [10].

Another recent example of reverse-engineering actually deals with Streebog. The linear layer of the permutation used to build its compression function was originally given as a binary matrix. However, it was shown in [27] that it corresponds to a matrix multiplication in $(\mathbb{F}_{2^8})^8$.

More generally, the task of reverse-engineering S-Boxes is related to finding generic attacks against high-level constructions. For instance, the cryptanalysis of SASAS [28], the recent attacks against the ASASA scheme [29,30] and the recovery of the secret Feistel functions for 5-, 6- and 7-round Feistel proposed in [31] can also be interpreted as methods to reverse-engineer S-Boxes built using such structures.

Our Contribution We managed to reverse-engineer the hidden structure of this S-Box. A simplified high level view is given in Figure 1. It relies on two rounds reminiscent of a Feistel or Misty-like structure where the output of the Feistel

ria and structure are public. In fact, the secrecy surrounding the S-Box of Streebog was part of the motivation behind this change [21].

function is combined with the other branch using a finite field multiplication. In each round, a different heuristic is used to prevent issues caused by multiplication by 0. This structure is hidden by two different whitening linear layers applied before and after it.

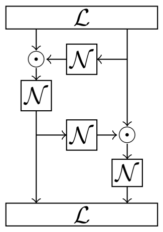


Fig. 1: A simplified view of our decomposition of π. Linear (resp. non linear) functions are denoted $\mathcal{L}$ (resp. $\mathcal{N}$) and $\odot$ is a finite field multiplication.

With the exception of the inverse function which is used once, none of the components of this decomposition exhibits particularly good cryptographic properties. In fact, one of the non-linear 4-bit permutations used has a probability 1 differential.

Our recovery of the structure of π relies on spotting visual patterns in its LAT and exploiting those. We generalize this method and show how visual patterns in the LAT of 4-round Feistel Networks can be exploited to decompose a so-called $\mathsf{AF}^4\mathsf{A}$ structure consisting in a 4-round Feistel Network whitened with affine layers.

Outline Section 2 introduces the definitions we need and describes how a statistical analysis of π rules out its randomness. Then, Section 3 explains the steps we used to reverse-engineer the S-Box starting from a picture derived from its linear properties and ending with a full specification of its secret structure. Section 4 is our analysis of the components used by GOST to build this S-Box. Finally, Section 5 describes a generic recovery attack against permutations affine-equivalent to 4-round Feistel Networks with secret components.

2 Boolean Functions and Randomness

2.1 Definitions and Notations

Definition 1. *We denote as $\mathbb{F}_p$ the finite field of size p. A* vectorial Boolean function *is a function mapping $\mathbb{F}_2^n$ to $\mathbb{F}_2^m$. We call* Boolean permutation *a permutation of $\mathbb{F}_2^n$.*

In what follows, we shall use the following operations and notations:

– exclusive-OR (or XOR) is denoted $\oplus$,

– logical AND is denoted $\wedge$,
– the scalar product of two elements $x = (x_{n-1}, ..., x_0)$ and $y = (y_{n-1}, ..., y_0)$ of $\mathbb{F}_2^n$ is denoted "$\cdot$" and is equal to to $x \cdot y = \bigoplus_{i=0}^{n-1} x_i \wedge y_i$, and
– finite field multiplication is denoted $\odot$.

The following tables are key tools to predict the resilience of an S-Box against linear and differential attacks.

DDT the *Difference Distribution Table* of a function f mapping n bits to m is a $2^n \times 2^m$ matrix $\mathcal{T}$ where $\mathcal{T}[\delta, \Delta] = \#\{x \in \mathbb{F}_2^n, f(x \oplus \delta) \oplus f(x) = \Delta\}$,

LAT the *Linear Approximation Table* of a function f mapping n bits to m is a $2^n \times 2^m$ matrix $\mathcal{L}$ where $\mathcal{L}[a, b] = \#\{x \in \mathbb{F}_2^n, a \cdot x = b \cdot f(x)\} - 2^{n-1}$. We note that coefficient $\mathcal{L}[a, b]$ can equivalently be expressed as follows:

$$\mathcal{L}[a, b] = \frac{1}{2} \sum_{x \in \mathbb{F}_2^n} (-1)^{a \cdot x \oplus b \cdot f(x)},$$

where the sum corresponds to the so-called *Walsh transform* of $x \mapsto (-1)^{b \cdot f(x)}$.

Furthermore, the coefficient $\mathcal{T}[\delta, \Delta]$ of a DDT $\mathcal{T}$ is called *cardinal of the differential* ($\delta \rightsquigarrow \Delta$) and the coefficient $\mathcal{L}[a, b]$ of a LAT $\mathcal{L}$ is called *bias of the approximation* ($a \rightsquigarrow b$).

From a designer's perspective, it is better to keep both the differential cardinals and the approximation biases low. For instance, the maximum cardinal of a differential is called the *differential uniformity* [5] and is chosen to be small in many primitives including the AES [1]. Such a strategy decreases the individual probability of all differential and linear trails.

Our analysis also requires some specific notations regarding linear functions mapping $\mathbb{F}_2^n$ to $\mathbb{F}_2^m$. Any such linear function can be represented by a matrix of elements in $\mathbb{F}_2$. For the sake of simplicity, we denote M^t the transpose of a matrix M and f^t the linear function obtained from the transpose of the matrix representation of the linear function f.

Finally, we recall the following definition.

Definition 2 (Affine-Equivalence). *Two vectorial Boolean functions f and g are affine-equivalent if there exist two affine mappings μ and η such that $f = \eta \circ g \circ \mu$.*

2.2 Quantifying Non-Randomness

In [10], Biryukov et al. proposed a general approach to try to reverse-engineer an S-Box suspected of having a hidden structure or of being built using secret design criteria. Part of their method allows a cryptanalyst to find out whether or not the S-Box could have been generated at random. It consists in checking if the distributions of the coefficients in both the DDT and the LAT are as it would be expected in the case of a random permutation.

The probability that all coefficients in the DDT of a random 8-bit permutation are at most equal to 8 and that this value occurs at most 25 times (as is the case for Streebog) is given by:

$$P[\max(d) = 8 \text{ and } N(8) \leq 25] \; = \; \sum_{\ell=0}^{25} \binom{255^2}{\ell} \cdot \Big[\sum_{d=0}^{3} \mathcal{D}(2d) \Big]^{255^2 - \ell} \cdot \mathcal{D}(8)^{\ell},$$

where $\mathcal{D}(d)$ is the probability that a coefficient of the DDT of a random permutation of $\mathbb{F}_2^8$ is equal to d. It is given in [32] and is equal to

$$\mathcal{D}(d) = \frac{e^{-1/2}}{2^{d/2}(d/2)!}.$$

We find that $P[\max(d) = 8 \text{ and } N(8) \leq 25] \approx 2^{-82.69}$. Therefore, we claim for π what Biryukov and Perrin claimed for the "F-Table" of Skipjack, namely that:

1. this S-Box was not picked uniformly at random from the set of the permutations of $\mathbb{F}_2^8$,
2. this S-Box was not generated by first picking many S-Boxes uniformly at random and then keeping the best according to some criteria, and
3. whatever algorithm was used to build it optimized the differential properties of the result.

3 Reverse-Engineering π

We used the algorithm described in [10] to try and recover possible structures for the S-Box. It has an even signature, meaning that it could be a Substitution-Permutation Network with simple bit permutations or a Feistel Network. However, the SASAS [28] attack and the SAT-based recovery attack against 3- ,4- and 5-round Feistel (both using exclusive-or and modular addition) from [10] failed. We also discarded the idea that π is affine-equivalent to a monomial of $\mathbb{F}_{2^8}$ using the following remark.

Remark 1. If f is affine-equivalent to a monomial, then every line of its DDT corresponding to a non-zero input difference contains the same coefficients (although usually in a different order).

This observation is an immediate consequence of the definition of the *differential spectrum* of monomials in [33]. For example, every line of the DDT of the S-Box of the AES, which is affine-equivalent to $x \mapsto x^{-1}$, contains exactly 129 zeroes, 126 twos and 1 four.

3.1 From a Vague Pattern to a Highly Structured Affine-Equivalent S-Box

It is also suggested in [10] to look at the so-called "Jackson Pollock representation" of the DDT and LAT of an unknown S-Box. These are obtained by assigning a color to each possible coefficient and drawing the table using one pixel per coefficient. The result for the absolute value of the coefficients of the LAT of π is given in Figure 2. While it may be hard to see on paper, blurry vertical lines appear when looking at a large enough version of this picture. In order to better see this pattern, we introduce the so-called $\oplus$-*texture*. It is a kind of auto-correlation.

Definition 3. *We call* $\oplus$-texture *of the LAT* $\mathcal{L}$ *of an S-Box the matrix* $\mathcal{T}^{\oplus}$ *with coefficients* $\mathcal{T}^{\oplus}[i, j]$ *defined as:*

$$\mathcal{T}^{\oplus}[i, j] \; = \; \#\big\{(x, y), |\mathcal{L}[x \oplus i, y \oplus j]| = |\mathcal{L}[x, y]|\big\}.$$

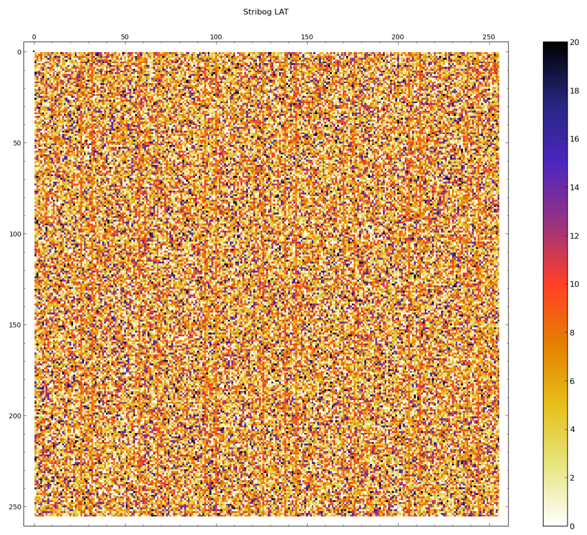


Fig. 2: The Jackson Pollock representation of the LAT of π (absolute value).

The Jackson Pollock representation of the $\oplus$-texture of the LAT $\mathcal{L}_{\pi}$ of π is given in Figure 3. The lines are now much more obvious and, furthermore, we observe dark dots in the very first column. The indices of both the rows containing the black dots and the columns containing the lines are the same and correspond to a vector space $\mathcal{V}$ where:

$$\mathcal{V} = \{00, 1a, 20, 3a, 44, 5e, 64, 7e, 8a, 90, aa, b0, ce, d4, ee, f4\}.$$

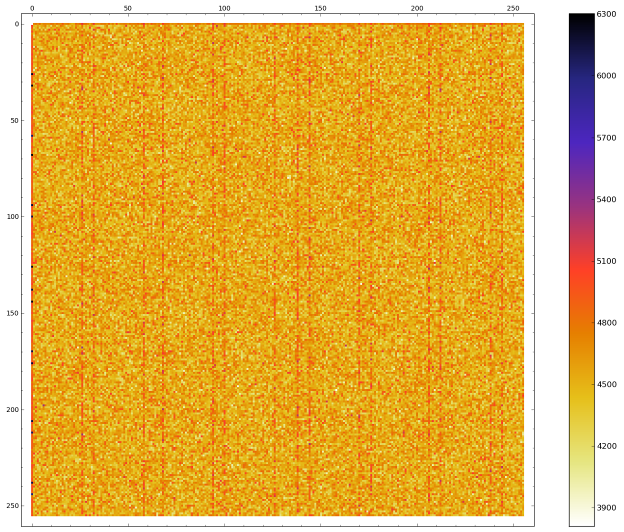


Fig. 3: The $\oplus$-texture of the LAT $\mathcal{L}_\pi$ of π.

In order to cluster the columns together to the left of the picture and the dark dots to the top of it, we can apply a linear mapping L to obtain a new table $\mathcal{L}'_\pi$ where $\mathcal{L}'_\pi[i,j] = \mathcal{L}_\pi[L(i), L(j)]$. We define L so that it maps $i \in \mathbb{F}_2^4$ to the i-th element of $\mathcal{V}$ and then complete it in a natural way to obtain a linear permutation of $\mathbb{F}_2^8$. It maps each bit as described below in hexadecimal notations:

$$L(01) = 1a, L(02) = 20, L(04) = 44, L(08) = 8a,$$

$$L(10) = 01, L(20) = 02, L(40) = 04, L(80) = 08.$$

The Jackson Pollock representation of $\mathcal{L}'_\pi$ is given in Figure 4. As we can see, it is highly structured: there is a 16×16 square containing[5] only coefficients equal to 0 in the top left corner. Furthermore, the left-most 15 bits to the right of column 0, exhibit a strange pattern: each of the coefficients in it has an absolute value in $[4, 12]$ although the maximum coefficient in the table is equal to 28. This forms a sort of low-contrast "stripe". The low number of different values it contains implies a low number of colour in the corresponding columns in $\mathcal{L}_\pi$, which in turn correspond to the lines we were able to distinguish in Figure 2.

It is natural to try and build another S-Box from π such that its LAT is equal to $\mathcal{L}'_\pi$. The remainder of this section describes how we achieved this. First, we describe a particular case[6] of Proposition 8.3 of [34] in the following lemma.

Lemma 1 (Proposition 8.3 of [34]). *Let f be a permutation mapping n bits to n and let $\mathcal{L}$ be its LAT. Let $\mathcal{L}'$ be a table defined by $\mathcal{L}'[u, v] = \mathcal{L}[\mu(u), v]$ for*

[5] Except of course in position $(0, 0)$ where the bias is equal to the maximum of 128.

[6] It is obtained by setting $b = b_0 = a = 0$ in the statement of the original proposition and renaming the functions used.

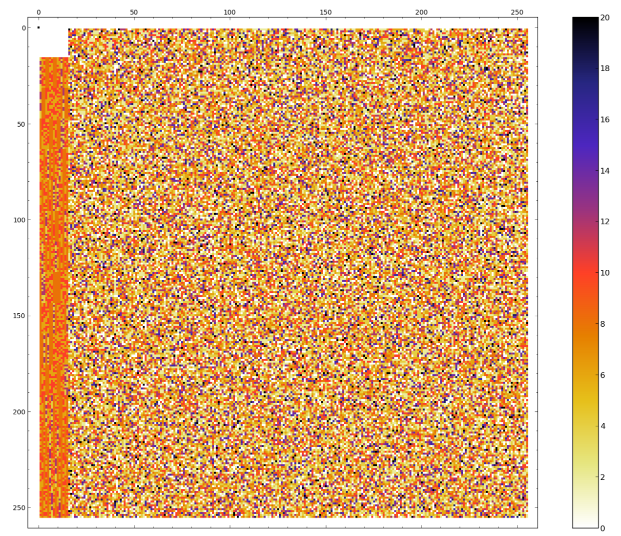


Fig. 4: The Jackson Pollock representation of $\mathcal{L}'_\pi$, where $\mathcal{L}'_\pi[i,j] = \mathcal{L}_\pi[L(i), L(j)]$.

some linear permutation μ. Then the function f' has LAT $\mathcal{L}'$, where

$$f' = f \circ (\mu^{-1})^t.$$

We also note that for a permutation f, the change of variable $y = f(x)$ implies:

$$\sum_{x \in \mathbb{F}_2^n} (-1)^{a \cdot x \oplus b \cdot f(x)} = \sum_{y \in \mathbb{F}_2^n} (-1)^{a \cdot f^{-1}(y) \oplus b \cdot y},$$

which in turn implies the following observation regarding the LAT of a permutation and its inverse.

Remark 2. Let f be a permutation mapping n bits to n and let $\mathcal{L}$ be its LAT. Then the LAT of its inverse f^{-1} is $\mathcal{L}^t$.

We can prove the following theorem using this remark and Lemma 1.

Theorem 1. *Let f be a permutation mapping n bits to n and let $\mathcal{L}$ be its LAT. Let $\mathcal{L}'$ be a table defined by $\mathcal{L}'[u,v] = \mathcal{L}[\mu(u), \eta(v)]$ for some linear permutations μ and η. Then the function f' has LAT $\mathcal{L}'$, where*

$$f' = \eta^t \circ f \circ (\mu^{-1})^t.$$

Proof. Let f be a permutation of n bits and let $\mathcal{L}$ be its LAT. We first build $f_\mu = f \circ (\mu^{-1})^t$ using Lemma 1 so that the LAT of f_μ is $\mathcal{L}_\mu$ with $\mathcal{L}_\mu[u,v] = \mathcal{L}[\mu(u), v]$. We then use Remark 2 to note that the inverse of f_μ has LAT $\mathcal{L}_\mu^{\text{inv}}$ with $\mathcal{L}_\mu^{\text{inv}}[u,v] = \mathcal{L}_\mu[v,u] = \mathcal{L}[v, \mu(u)]$. Thus, $f_\eta = f_\mu^{-1} \circ (\eta^{-1})^t$ has LAT $\mathcal{L}_\eta$ with $\mathcal{L}_\eta[u,v] = \mathcal{L}[\eta(v), \mu(u)]$. Using again Remark 2, we obtain that $f_\eta^{-1} = \eta^t \circ f \circ (\mu^{-1})^t$ has LAT $\mathcal{L}'$. $\square$

As a consequence of Theorem 1, the S-Box $L^t \circ \pi \circ (L^t)^{-1}$ has $\mathcal{L}'_\pi$ as its LAT. The mapping L^t consists in a linear Feistel round followed by a permutation of the left and right 4-bit nibbles (which we denote $\mathsf{swapNibbles}$). To simplify the modifications we make, we remove the nibble permutation and define

$$\pi' = L^* \circ \pi \circ L^*$$

where L^* is the Feistel round in L^t and is described in Figure 5.

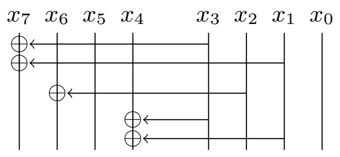


Fig. 5: A circuit computing L^* where its input is given in binary.

3.2 The First Decomposition

This affine-equivalent S-Box π' is highly structured. First of all, the LAT of $\big(\mathsf{swapNibbles} \circ \pi' \circ \mathsf{swapNibbles}\big)$ is $\mathcal{L}'_\pi$, with its white square in the top left and strange left side. It also has interesting multiset properties. We use notations similar to those in [28], i.e.:

C denotes a 4-bit nibble which is constant,
? denotes a set with no particular structure, and
P denotes a 4-bit nibble taking all 16 values.

Table 1 summarizes the multiset properties of π' and π'^{-1}. As we can see, these are similar to those of a 3-rounds Feistel. However, using the SAT-based algorithm from [10], we ruled out this possibility.

π'	input	output
	(P, C)	$(?, ?)$
	(C, P)	$(P, ?)$

π'^{-1}	input	output
	(P, C)	$(?, ?)$
	(C, P)	$(P, ?)$

Table 1: The multiset properties of π' and its inverse.

When looking at the inverse of π', we notice that the multiset property is actually even stronger. Indeed, for any constant ℓ, the set $\mathcal{S}_\ell = \{\pi'^{-1}(\ell||r), \forall r \in [0, 15]\}$ is almost a vector space. If we replace the unique element of $\mathcal{S}_\ell$ of the form $(?||0)$ by $(0||0)$, the set obtained is a vector space V_ℓ where the right nibble is a linear function of the left nibble. As stated before, the left nibble takes all possible values. If we put aside the outputs of the form $(?||0)$ then π'^{-1} can be seen as

$$\pi'^{-1}(\ell||r) = T_\ell(r)||V_\ell\big(T_\ell(r)\big),$$

In this decomposition, T is a 4-bit block cipher with a 4-bit key where the left input of π'^{-1} acts as a key. On the other hand, V is a keyed linear function: for all ℓ, V_ℓ is a linear function mapping 4 bits to 4 bits.

We then complete this alternative representation by replacing $V_\ell(0)$, which should be equal to 0, by the left side of $\pi'^{-1}(\ell||T_\ell^{-1}(0))$. This allows to find a high level decomposition of π'^{-1}.

Finally, we define a new keyed function $U_r(\ell) = V_\ell(r)$ and notice that, for all r, U_r is a permutation. A decomposition of π'^{-1} is thus:

$$\pi'^{-1}(\ell||r) = T_\ell(r)||U_{T_\ell(r)}(\ell),$$

where the full codebooks of both mini-block ciphers T and U are given in the Appendix in Tables 12a and 12b and respectively. This structure is summarized in Figure 6.

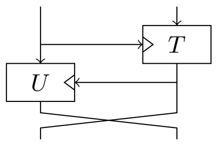


Fig. 6: The high level structure of π'^{-1}.

We decompose the mini-block ciphers T and U themselves in Section 3.3 and Section 3.4 respectively.

3.3 Reverse-Engineering T

We note that the mini-block cipher T' defined as $T'_k : x \mapsto T_k\big(x \oplus t_{\text{in}}(k) \oplus 0xC\big)$ for $t_{\text{in}}(k) = 0||k_2||k_3||0$ (see Table 2) is such that $T'_k(0) = 0$ for all k.

Furthermore, T' is such that all lines of T'_k can be obtained through a linear combination of T'_6, T'_7, T'_8 and T'_9 as follows:

$$\begin{array}{lll} T'_0 = T'_7 \oplus T'_9 & T'_1 = T'_8 \oplus T'_9 & T'_2 = T'_7 \oplus T'_9 \\ T'_3 = T'_6 \oplus T'_7 & T'_4 = T'_7 & T'_5 = T'_7 \oplus T'_8 \\ T'_a = T'_6 \oplus T'_7 \oplus T'_8 \oplus T'_9 & T'_b = T'_6 \oplus T'_7 & T'_c = T'_6 \oplus T'_7 \oplus T'_8 \\ T'_d = T'_9 & T'_e = T'_8 & T'_f = T'_7 \oplus T'_9. \end{array} \quad (1)$$

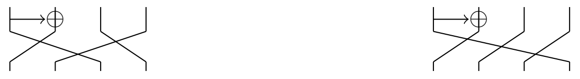
We also notice that T'_6, T'_7, T'_8 and T'_9 are all affine equivalent. Indeed, the linear mapping A defined by $A : 1 \mapsto 4, 2 \mapsto 1, 4 \mapsto 8, 8 \mapsto a$ (see Figure 7a) is such that:

$$\begin{aligned} T'_7 &= A \circ T'_6 \\ T'_8 &= A^2 \circ T'_6 \\ T'_9 &= A^3 \circ T'_6. \end{aligned}$$

If we swap the two least significant bits (an operation we denote swap2lsb) before and after applying A we see a clear LFSR structure (see Figure 7b).

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
6	0	2	7	c	4	a	5	6	3	9	d	f	1	e	b	8
7	0	1	d	2	8	b	c	9	5	e	6	7	4	3	f	a
8	0	4	6	1	a	f	2	e	c	3	9	d	8	5	7	b
9	0	8	9	4	b	7	1	3	2	5	e	6	a	c	d	f
0	0	9	4	6	3	c	d	a	7	b	8	1	e	f	2	5
1	0	c	f	5	1	8	3	d	e	6	7	b	2	9	a	4
2	0	9	4	6	3	c	d	a	7	b	8	1	e	f	2	5
3	0	3	a	e	c	1	9	f	6	7	b	8	5	d	4	2
4	0	1	d	2	8	b	c	9	5	e	6	7	4	3	f	a
5	0	5	b	3	2	4	e	7	9	d	f	a	c	6	8	1
a	0	f	5	b	d	9	a	2	8	1	c	3	7	4	e	6
b	0	3	a	e	c	1	9	f	6	7	b	8	5	d	4	2
c	0	7	c	f	6	e	b	1	a	4	2	5	d	8	3	9
d	0	8	9	4	b	7	1	3	2	5	e	6	a	c	d	f
e	0	4	6	1	a	f	2	e	c	3	9	d	8	5	7	b
f	0	9	4	6	3	c	d	a	7	b	8	1	e	f	2	5

Table 2: A modified version T' of the mini-block cipher T.



(a) Definition of A. (b) Swapping the least significant bits.

Fig. 7: The mapping used to generate T'_7, T'_8 and T'_9 from T'_6.

We deduce the LFSR polynomial to be $X^4 + X^3 + 1$. This points towards finite field multiplication and, indeed, the mapping $\hat{A} = \mathsf{swap2lsb} \circ A \circ \mathsf{swap2lsb}$ can be viewed as a multiplication by X in $\mathbb{F}_{2^4} = \mathbb{F}_2[X]/(X^4+X^3+1)$. To fit the swap into the original scheme we modify T'_6 and the bottom linear layer. Indeed, note that

$$A^i = (\mathsf{swap2lsb} \circ \hat{A} \circ \mathsf{swap2lsb})^i = \mathsf{swap2lsb} \circ \hat{A}^i \circ \mathsf{swap2lsb} \quad \text{for } i = 0, 1, \ldots,$$

so that we can merge one $\mathsf{swap2lsb}$ into T'_6 and move the other $\mathsf{swap2lsb}$ through XOR's outside T'. Let $t = \mathsf{swap2lsb} \circ T'_6$. Then $\mathsf{swap2lsb} \circ T'_k(x)$ is a linear combination of $X^i \odot t(x)$, where $i \in \{0, 1, 2, 3\}$ and $\odot$ is multiplication in the specified field. Thus, T can be computed as follows:

$$T_k(x) = \mathsf{swap2lsb}\left(f(k) \odot t\big(x \oplus t_{\text{in}}(k) \oplus 0xC\big)\right),$$

where f captures the linear relations from Equations (1). Both f and t are given in Table 3 and a picture representing the structure of T is given in Figure 8.

Note that $f(x)$ is never equal to 0: if it were the case then the function would not be invertible. On the other hand, the inverse of T_k is easy to compute: f must be replaced by $1/f$ where the inversion is done in the finite field $\mathbb{F}_{2^4}$, t by its inverse t^{-1} and the order of the operations must be reversed.

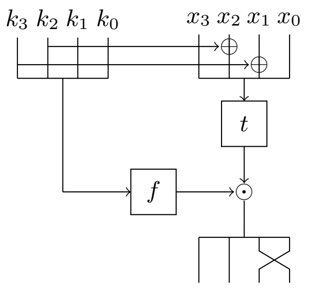


Fig. 8: The mini-block cipher T.

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
f	a	c	a	3	2	6	1	2	4	8	f	3	7	8	4	a
t	2	d	b	8	3	a	e	f	4	9	6	5	0	1	7	c

Table 3: Mappings f and t.

3.4 Reverse-Engineering U

Since $U_k(x) = V_x(k)$ and V_x is a linear function when $x \neq 0$, we have

$$U_k(x) = \big(k_3 \times U_8(x)\big) \oplus \big(k_2 \times U_4(x)\big) \oplus \big(k_1 \times U_2(x)\big) \oplus \big(k_0 \times U_1(x)\big)$$

where $k = \sum_{i \leq 3} k_i 2^i$ and $k \neq 0$. We furthermore notice that the permutations U_2, U_4 and U_8 can all be derived from U_1 using some affine functions B_k so that $U_k = B_k \circ U_1$. The values of $B_k(x)$ are given in Table 4.

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
B_2	5	c	0	9	2	b	7	e	3	a	6	f	4	d	1	8
B_4	1	d	7	b	f	3	9	5	c	0	a	6	2	e	4	8
B_8	5	6	d	e	0	3	8	b	a	9	2	1	f	c	7	4

Table 4: Affine functions such that $U_k = B_k \circ U_1$.

If we let $B(x) = B_4(x) \oplus 1$ then $B_2(x) = B^{-1}(x) \oplus 5$ and $B_8(x) = B^2(x) \oplus 5$. Thus, we can define a linear function u_{out} such that

$$\begin{aligned} U_1(x) &= B^0 \circ U_1(x) \oplus u_{\text{out}}(1) \\ U_2(x) &= B^{-1} \circ U_1(x) \oplus u_{\text{out}}(2) \\ U_4(x) &= B^1 \circ U_1(x) \oplus u_{\text{out}}(4) \\ U_8(x) &= B^2 \circ U_1(x) \oplus u_{\text{out}}(8). \end{aligned} \tag{2}$$

Let M_2 be the matrix representation of multiplication by X in the finite field we used to decompose T, namely $\mathbb{F}_{2^4} = \mathbb{F}_2[X]/(X^4 + X^3 + 1)$. The linear mapping u_f defined by $u_f : 1 \mapsto 5, 2 \mapsto 2, 4 \mapsto 6, 8 \mapsto 8$ is such that $B = u_f \circ M_2 \circ u_f^{-1}$ is so that Equations (2) can be re-written as

$$\begin{aligned} U_1(x) &= u_f \circ M_2^0 \circ u_f^{-1} \circ U_1(x) \oplus u_{\text{out}}(1) \\ U_2(x) &= u_f \circ M_2^{-1} \circ u_f^{-1} \circ U_1(x) \oplus u_{\text{out}}(2) \\ U_4(x) &= u_f \circ M_2^1 \circ u_f^{-1} \circ U_1(x) \oplus u_{\text{out}}(4) \\ U_8(x) &= u_f \circ M_2^2 \circ u_f^{-1} \circ U_1(x) \oplus u_{\text{out}}(8). \end{aligned} \tag{3}$$

If we swap the two least significant bits of k, then the exponents of matrix M_2 will go in ascending order: $(-1, 0, 1, 2)$. Let $u_1 = M_2^{-1} \circ u_f^{-1} \circ U_1(x)$. Since M_2 is multiplication by X in the finite field, we can write the following expression for U_k (when $k \neq 0$):

$$U_k(x) = u_f\big(u_1(x) \odot \mathsf{swap2lsb}(k)\big) \oplus u_{out}(k). \tag{4}$$

The complete decomposition of U is presented in Figure 9. It uses the 4-bit permutations u_0 and u_1 specified in Table 5. We could not find a relation between u_1 and $u_0 = u_f^{-1} \circ U_0(x)$ so there has to be a conditional branching: U selects the result of Equation (4) if $k \neq 0$ and the result of $u_0(x)$ otherwise before applying u_f. This is achieved using a multiplexer which returns the output of u_0 if $k_3 = k_2 = k_1 = k_0 = 0$, and returns the output of u_1 if it is not the case. In other words, U can be computed as follows:

$$U_k(x) = \begin{cases} u_f\big(u_1(x) \odot \mathsf{swap2lsb}(k)\big) \oplus u_{\text{out}}(k), & \text{if } k \neq 0 \\ u_f\big(u_0(x)\big) & \text{if } k = 0. \end{cases}$$

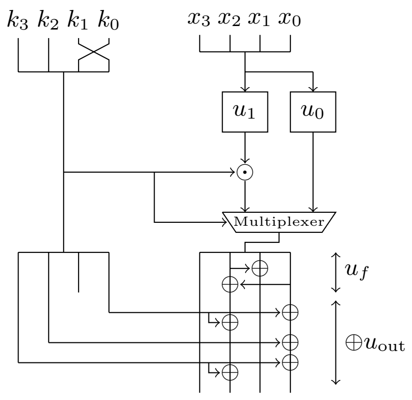


Fig. 9: The mini-block cipher U.

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
u_0	8	b	0	2	9	1	4	f	c	5	7	3	e	d	6	a
u_1	4	7	d	e	8	9	1	0	6	3	f	a	2	c	b	5

Table 5: Permutations u_0 and u_1.

3.5 The Structure of π

In Sections 3.3 and 3.4, we decomposed the two mini-block ciphers T and U which can be used to build π'^{-1}, the inverse of $L^* \circ \pi \circ L^*$. These mini-block ciphers are based on the non-linear 4-bit functions f, t, u_0, u_1, two finite field multiplications, a "trick" to bypass the non-invertibility of multiplication by 0 and simple linear functions. Let us now use the expressions we identified to express π itself.

First, we associate the linear functions and L^* into α and ω, two linear permutations applied respectively at the beginning and the end of the computation.

α First of all, we need to apply L^* as well as the the swap of the left and right branches (swapNibbles) present in the high level decomposition of π'^{-1} (see Figure 6). Then, we note that the key in U needs a swap of its 2 bits of lowest weight (swap2lsb) and that the ciphertext of T needs the same swap. Thus, we simply apply swap2lsb. Then, we apply the addition of u_{out} and the inverse of u_f.

ω This function is simpler: it is the composition of the addition of t_{in} and of L^*.

The matrix representations of these layers are

$$\alpha = \begin{bmatrix} 0&0&0&0&1&0&0&0 \\ 0&1&0&0&0&0&0&1 \\ 0&1&0&0&0&0&1&1 \\ 1&1&1&0&1&1&1&1 \\ 1&0&0&0&1&0&1&0 \\ 0&1&0&0&0&1&0&0 \\ 0&0&0&1&1&0&1&0 \\ 0&0&1&0&0&0&0&0 \end{bmatrix}, \ \omega = \begin{bmatrix} 0&0&0&0&1&0&1&0 \\ 0&0&0&0&0&1&0&0 \\ 0&0&1&0&0&0&0&0 \\ 1&0&0&1&1&0&1&0 \\ 0&0&0&0&1&0&0&0 \\ 0&1&0&0&0&1&0&0 \\ 1&0&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&1 \end{bmatrix}.$$

In order to invert U, we define $\nu_0 = u_0^{-1}$ and $\nu_1 = u_1^{-1}$. If $\ell = 0$, then the output of the inverse of U is $\nu_0(r)$, otherwise it is $\nu_1\big(r \odot \mathcal{I}(\ell)\big)$, where $\mathcal{I} : x \mapsto x^{14}$ is the multiplicative inverse in $\mathbb{F}_{2^4}$. To invert T, we define $\sigma = t^{-1}$ and $\phi = \mathcal{I} \circ f$ and compute $\sigma\big(\phi(\ell) \odot r\big)$.

Figure 10 summarizes how to compute π using these components. The non-linear functions are all given in Table 6. We also provide a SAGE [35] script performing those computations in Appendix E. It can also be downloaded from Github.[7] The evaluation of $\pi(\ell||r)$ can be done as follows:

```
1. $(\ell||r) := \alpha(\ell||r)$
2. if $r = 0$ then $\ell := \nu_0(\ell)$, else $\ell := \nu_1\big(\ell \odot \mathcal{I}(r)\big)$
3. $r := \sigma\big(r \odot \phi(l)\big)$
4. return $\omega(\ell||r)$.
```

4 Studying the Decomposition of π

4.1 Analyzing the Components

Table 7 summarizes the properties of the non-linear components of our decomposition. While it is not hard to find 4-bit permutations with a differential uniformity of 4, we see that none of the components chosen do except for the inverse function. We can thus discard the idea that the strength of π against differential and linear attacks relies on the individual resilience of each of its components.

As can be seen in Table 7, there is a probability 1 differential in ν_1: $9 \rightsquigarrow 2$. Furthermore, a difference equal to 2 on the left branch corresponds to a 1 bit difference on bit 5 of the input of ω, a bit which is left unchanged by ω.

[7] https://github.com/picarresursix/GOST-pi

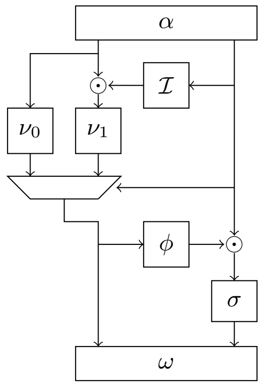


Fig. 10: Our decomposition of π.

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
$\mathcal{I}$	0	1	c	8	6	f	4	e	3	d	b	a	2	9	7	5
ν_0	2	5	3	b	6	9	e	a	0	4	f	1	8	d	c	7
ν_1	7	6	c	9	0	f	8	1	4	5	b	e	d	2	3	a
ϕ	b	2	b	8	c	4	1	c	6	3	5	8	e	3	6	b
σ	c	d	0	4	8	b	a	e	3	9	5	2	f	1	6	7

Table 6: The non-linear functions needed to compute π.

	1-to-1	Best differentials and their probabilities	Best linear approximations and their probabilities
ϕ	No	`1` $\rightsquigarrow$ `d` (8/16)	`3` $\rightsquigarrow$ `8` (2/16), `7` $\rightsquigarrow$ `d` (2/16)
σ	Yes	`f` $\rightsquigarrow$ `b` (6/16)	`1` $\rightsquigarrow$ `f` (14/16)
ν_0	Yes	`6` $\rightsquigarrow$ `c` (6/16), `e` $\rightsquigarrow$ `e` (6/16)	30 approximations $(8 \pm 4)/16$
ν_1	Yes	`9` $\rightsquigarrow$ `2` (16/16)	8 approximations $(8 \pm 6)/16$

Table 7: Linear and differential properties of the components of π.

The structure itself also implies the existence of a truncated differential with high probability. Indeed, if the value on the left branch is equal to 0 for two different inputs, then the output difference on the left branch will remain equal to 0 with probability 1. This explains why the probability that a difference in $\Delta_{\text{in}} = \{\alpha^{-1}(\ell||0), \ell \in \mathbb{F}_2^4, x \neq 0\}$ is mapped to a difference in $\Delta_{\text{out}} = \{\omega(\ell||0), \ell \in \mathbb{F}_2^4, x \neq 0\}$ is higher than the expected 2^{-4}:

$$\frac{1}{2^4-1} \sum_{\delta \in \Delta_{\text{in}}} P[\pi(x \oplus \delta) \oplus \pi(x) \in \Delta_{\text{out}}] = \frac{450}{(2^4-1) \times 2^8} \approx 2^{-3}.$$

4.2 Comments on the Structure Used

We define $\hat{\pi}$ as $\omega^{-1} \circ \pi \circ \alpha^{-1}$, i.e. π minus its whitening linear layers.

The structure of $\hat{\pi}$ is similar to a 2-round combination of a Misty-like and Feistel structure where the XORs have been replaced by finite field multiplications. To the best of our knowledge, this is the first time such a structure has been used in cryptography. Sophisticated lightweight decompositions of the S-Box of the AES rely on finite field multiplication in $\mathbb{F}_{2^4}$, for instance in [36]. However, the high level structure used in this case is quite different. If π corresponds to such a decomposition then we could not find what it corresponds to. Recall in particular that π cannot be affine-equivalent to a monomial.

The use of finite field multiplication in such a structure yields a problem: if the output of the "Feistel function" is equal to 0 then the structure is not invertible. This issue is solved in a different way in each round. During the first round, a different data-path is used in the case which should correspond to a multiplication by zero. In the second round, the "Feistel function" is not bijective and, in particular, has no pre-image for 0.

Our decomposition also explains the pattern in the LAT[8] of π and π' that we used in Section 3.1 to partially recover the linear layers permutations α and ω. This pattern is made of two parts: the white square appearing at the top-left of $\mathcal{L}'_\pi$ and the "stripe" covering the 16 left-most columns of this table (see Figure 4). In what follows, we explain why the white square and the stripe are present in $\mathcal{L}'_\pi$. We also present an alternative representation of $\hat{\pi}$ which highlights the role of the multiplexer.

On the White Square We first define a *balanced function* and the concept of *integral distinguishers*. Using those, we can rephrase a result from [37] as Lemma 2.

Definition 4 (Balanced Function). *Let $f : \mathbb{F}_2^n \to \mathbb{F}_2^m$ be a Boolean function. We say that f is* balanced *if the size of the preimage $\{x \in \mathbb{F}_2^n, f(x) = y\}$ of y is the same for all y in $\mathbb{F}_2^m$.*

Definition 5 (Integral Distinguisher). *Let f be a Boolean function mapping n bits to m. An integral distinguisher consists in two subsets $\mathcal{C}_{in} \subseteq [0, n-1]$ and $\mathcal{B}_{out} \subseteq [0, m-1]$ of input and output bit indices with the following property: for all c, the sum $\bigoplus_{x \in \mathcal{X}} f(x)$ restricted to the bits with indices in $\mathcal{B}_{out}$ is balanced; where $\mathcal{X}$ is the set containing all x such that the bits with indices in $\mathcal{C}_{in}$ are fixed to the corresponding value in the binary expansion of c (so that $|\mathcal{X}| = 2^{n-|\mathcal{C}_{in}|}$).*

Lemma 2 ([37]). *Let f be a Boolean function mapping n bits to m and with LAT $\mathcal{L}_f$ for which there exists an integral distinguisher $(\mathcal{C}_{in}, \mathcal{B}_{out})$. Then, for all (a, b) such that the 1 in the binary expansion of a all have indices in $\mathcal{C}_{in}$ and the 1 in the binary expansion of b have indices in $\mathcal{B}_{out}$, it holds that $\mathcal{L}_f[a, b] = 0$.*

This theorem explains the presence of the white square in $\mathcal{L}'_\pi$. Indeed, fixing the input of the right branch of $\hat{\pi}$ leads to a permutation of the left branch in the plaintext becoming a permutation of the left branch in the ciphertext; hence the existence of an integral distinguisher for $\hat{\pi}$ which in turn explains the white square.

[8] Note that the LAT of $\hat{\pi}$ is not exactly the same as $\mathcal{L}'_\pi$ which is given in Figure 4 because e.g. of a nibble swap.

On the Stripe Biases in the stripe correspond to approximations $(a_L||a_R \rightsquigarrow b_L||0)$ in $\hat{\pi}$, where $b_L > 0$. We detail the computation of the corresponding biases in Appendix D. It turns out that the expression of $\mathcal{L}[a_L||a_R, b_L||0]$ is

$$\mathcal{L}[a_L||a_R, b_L||0] = \mathcal{L}_0[a_L, b_L] + 8 \times \left((-1)^{b_L \cdot y_0} - \hat{\delta}(b_L)\right),$$

where $\mathcal{L}_0$ is the LAT of ν_0, y_0 depends on a_R, a_L and the LAT of ν_1, and $\hat{\delta}(b_L)$ is equal to 1 if $b_L = 0$ and to 0 otherwise. Very roughly, ν_1 is responsible for the sign of the biases in the stripe and ν_0 for their values.

Since the minimum and maximum biases in $\mathcal{L}^0$ are -4 and $+4$, the absolute value of $\mathcal{L}[a_L||a_R, b_L||0]$ is indeed in $[4, 12]$. As we deduce from our computation of these biases, the stripe is caused by the conjunction of three elements:

- the use of a multiplexer,
- the use of finite field inversion, and
- the fact that ν_0 has good non-linearity.

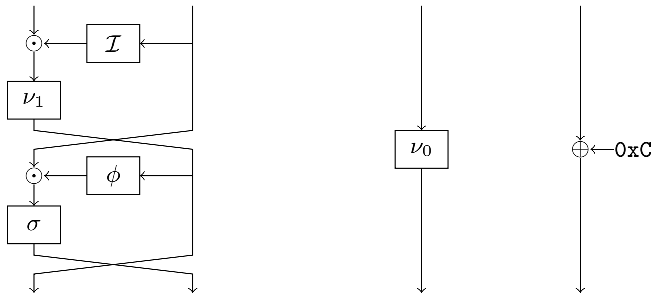
Ironically, the only "unsurprising" sub-component of π, namely the inverse function, is one of the reasons why we were able to reverse-engineer this S-Box in the first place. Had $\mathcal{I}$ been replaced by a different (and possibly weaker!) S-Box, there would not have been any of the lines in the LAT which got our reverse-engineering started. Note however that the algorithm based on identifying linear subspaces of zeroes in the LAT of a permutation described in Section 5 would still work because of the white-square.

Alternative Representation Because of the multiplexer, we can deduce an alternative representation of $\hat{\pi}$. If the right nibble of the input is not equal to 0 then $\hat{\pi}$ can be represented as shown in Figure 11a. Otherwise, it is essentially equivalent to one call to the 4-bit S-Box ν_0, as shown in Figure 11b.

Note also that the decomposition we found is not unique. In fact, we can create many equivalent decompositions by e.g. adding multiplication and division by constants around the two finite field multiplications. We can also change the finite field in which the operations are made at the cost of appropriate linear isomorphisms modifying the 4-bit S-Boxes and the whitening linear layers. However the presented decomposition is the most structured that we have found.

4.3 Hardware Implementation

It is not uncommon for cryptographers to build an S-Box from smaller ones, typically an 8-bit S-Box from several 4-bit S-Boxes. For example, S-Boxes used in Whirlpool [20], Zorro [38], Iceberg [39], Khazad [40], CLEFIA [41], and Robin and Fantomas [42] are permutations of 8 bits based on smaller S-Boxes. In many cases, such a structure is used to allow an efficient implementation of the S-Box in hardware or using a bit-sliced approach. In fact, a recent work by Canteaut et al. [43] focused on how to build efficient 8-bit S-Boxes from 3-round Feistel and Misty-like structures. Another possible reason behind such a choice is given



(a) $\hat{\pi}$ when $r \neq 0$. (b) $\hat{\pi}$ when $r = 0$.

Fig. 11: An alternative representation of $\hat{\pi}$ where $\pi = \omega \circ \hat{\pi} \circ \alpha$.

by the designers e.g. of CLEFIA: it is to prevent attacks based on the algebraic properties of the S-Box, especially if it is based on the inverse in $\mathbb{F}_{2^8}$ like in the AES. Finally, a special structure can help achieve special properties. For instance, the S-Box of Iceberg is an involution obtained using smaller 4-bit involutions and a Substitution-Permutation Network.

As stated in the introduction, hardware optimization was supposed to be one of the design criteria used by the designers of π. Thus, it is reasonable to assume that one of the aims of the decomposition we found was to decrease the hardware footprint of the S-Box.

To test this hypothesis, we simulated the implementation of π in hardware.[9] We used three different definitions of π: the look up table given by the designers, our decomposition and, finally, a tweaked decomposition where the multiplexer is moved lower.[10] Table 8 contains both the area taken by our implementations and the delay, i.e. the time taken to compute the output of the S-Box. For both quantities, the lower is the better. As we can see, the area is divided by up to 2.5 and the delay by 8, meaning that an implementer knowing the decomposition has a significant advantage over one that does not.

Structure	Area (μm^2)	Delay (ns)
naive implementation	3889.6	362.52
using the decomposition	1534.7	61.53
decomposition and tweaked MUX	1530.1	46.11

Table 8: Results on the hardware implementation of π.

[9] We used `Synopsys design_compiler` (version J-2014.09-SP2) along with digital library `SAED_EDK90_CORE` (version 1.11).

[10] More precisely, the multiplexer is moved after the left side is input to ϕ. This does not change the output: when the output of ν_0 is selected, the right branch is equal to 0 and the input of σ is thus 0 regardless of the left side.

5 Another LAT-Based Attack Against Linear Whitening

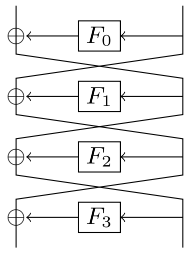
Our attack against π worked by identifying patterns in a visual representation of its LAT and exploiting them to recover parts of the whitening linear layers surrounding the core of the permutation.

It is possible to exploit other sophisticated patterns in the LAT of a permutation. In the remainder of this section, we describe a specific pattern in the LAT of a 4-round Feistel Network using bijective Feistel functions. We then use this pattern in conjunction with Theorem 1 to attack the $\mathsf{AF}^4\mathsf{A}$ structure corresponding to a 4-round Feistel Network with whitening linear layers. Note that more generic patterns such as white rectangles caused by integral distinguishers (see Section 4.2) could be used in a similar fashion to attack other generic constructions, as we illustrate in Section 5.3. The attack principle is always the same:

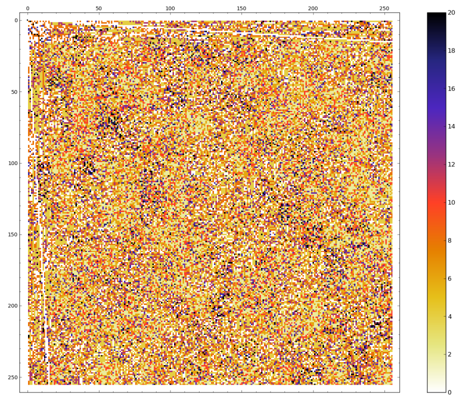
1. identify patterns in the LAT,
2. deduce partial whitening linear layers,
3. recover the core of the permutation with an *ad hoc* attack.

5.1 Patterns in the LAT of a 4-Round Feistel Network

Let $F_0, ..., F_3$ be four n-bit Boolean permutations. Figure 12a represents the 4-round Feistel Network f built using F_i as its i-th Feistel function. Figure 12b is the Pollock representation of the LAT of a 8-bit Feistel Network f_{exp} built using four 4-bit permutations picked uniformly at random.



(a) Definition of f.



(b) The LAT (Pollock repr.) of f_{exp}.

Fig. 12: A 4-round Feistel Network and its LAT.

In Figure 12b, we note that the LAT $\mathcal{L}_{\text{exp}}$ of f_{exp} contains both vertical and horizontal segments of length 16 which are made only of zeroes. These segments form two lines starting at (0,0), one ending at (15,255) and another one ending in (255,15), where (0,0) is the top left corner. The vertical segments are in columns 0 to 15 and correspond to entries $\mathcal{L}_{\text{exp}}[a_L||a_R, 0||a_L]$ for any (a_L, a_R). The horizontal ones are in lines 0 to 15 and correspond to entries $\mathcal{L}_{\text{exp}}[0||a_R, a_R||b_L]$ for any (a_L, a_R).

Let us compute the coefficients which correspond to such vertical segments for any 4-round Feistel Network f with LAT $\mathcal{L}$. These are equal to

$$\begin{aligned}\mathcal{L}[a_L||a_R, 0||a_L] &= \sum_{x \in \mathbb{F}_2^{2n}} (-1)^{(a_L||a_R)\cdot x \oplus a_L \cdot f(x)} \\ &= \sum_{r \in \mathbb{F}_2^n} (-1)^{a_R \cdot r} \sum_{\ell \in \mathbb{F}_2^n} (-1)^{a_L \cdot \left(\ell \oplus f_R(\ell||r)\right)},\end{aligned}$$

where $f_R(x)$ is the right word of $f(x)$. This quantity is equal to $\ell \oplus F_0(r) \oplus F_2\big(r \oplus F_1(\ell \oplus F_0(r))\big)$, so that $\mathcal{L}[a_L||a_R, 0||a_L]$ can be re-written as:

$$\mathcal{L}[a_L||a_R, 0||a_L] = \sum_{r \in \mathbb{F}_2^n} (-1)^{a_R \cdot r} \sum_{\ell \in \mathbb{F}_2^n} (-1)^{a_L \cdot \left(F_0(r) \oplus F_2\left(r \oplus F_1(\ell \oplus F_0(r))\right)\right)}.$$

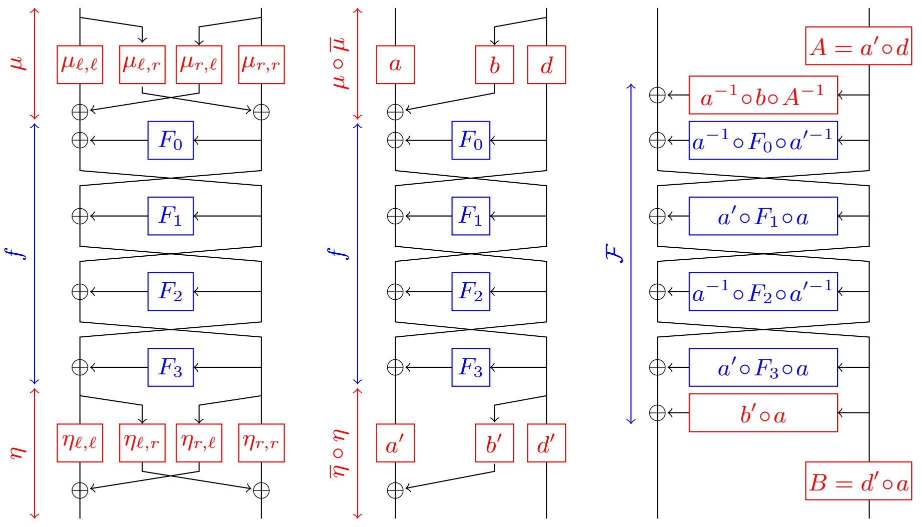
Since $\ell \mapsto F_2\big(r \oplus F_1(\ell \oplus F_0(r))\big)$ is balanced for all r, the sum over ℓ is equal to 0 for all r (unless $a_L = 0$). This explains[11] the existence of the vertical "white segments". The existence of the horizontal ones is a simple consequence of Remark 2: as the inverse of f is also a 4-round Feistel, its LAT must contain white vertical segments. Since the LAT of f is the transpose of the LAT of f^{-1}, these vertical white segments become the horizontal ones.

5.2 A Recovery Attack Against AF^{4}A

These patterns can be used to attack a 4-round Feistel Network whitened using affine layers, a structure we call AF^{4}A. Applying the affine layers before and after a 4-round Feistel Network scrambles the white segments in the LAT in a linear fashion - each such segment becomes an affine subspace. The core idea of our attack is to compute the LAT of the target and then try to rebuild both the horizontal and vertical segments. In the process, we will recover parts of the linear permutations applied to the rows and columns of the LAT of the inner Feistel Network and, using Theorem 1, recover parts of the actual linear layers. Then the resulting 4-round Feistel Network can be attacked using results presented in [31]. By parts of a linear layer we understand the four $(n \times n)$-bit submatrices of the corresponding matrix.

First Step: Using the LAT Let $f : \mathbb{F}_2^{2n} \to \mathbb{F}_2^{2n}$ be a 4-round Feistel Network and let $g = \eta \circ f \circ \mu$ be its composition with some whitening linear layers η and μ. The structure of g is presented in Figure 13a using $(n \times n)$-bit matrices $\mu_{\ell,\ell}, \mu_{r,\ell}, \mu_{\ell,r}, \mu_{r,r}$ for μ and $\eta_{\ell,\ell}, \eta_{r,\ell}, \eta_{\ell,r}, \eta_{r,r}$ for η.

[11] Note that our proof actually only requires F_1 and F_2 to be permutations. The pattern would still be present if the first and/or last Feistel functions had inner-collisions.



(a) $g = \eta \circ f \circ \mu$. (b) $g'' = \overline{\eta} \circ g \circ \overline{\mu}$. (c) g'' (alt. representation).

Fig. 13: The effect of $\overline{\mu}$ and $\overline{\eta}$ on g. Linear layers are in red and inner Feistel Networks in blue.

Assume that we have the full codebook of g and therefore that we can compute the LAT $\mathcal{L}_g$ of g. By Theorem 1, it holds that $\mathcal{L}_g[u, v] = \mathcal{L}_f[(\mu^{-1})^t(u), \eta^t(v)]$ and, equivalently, that $\mathcal{L}_g[\mu^t(u), (\eta^{-1})^t(v)] = \mathcal{L}_f[u, v]$.

We use Algorithm 1 (see next section) to find a linear subspace $\mathcal{S}$ of $\mathbb{F}_2^{2n}$ such that $|\mathcal{S}| = 2^n$ and such that it has the following property: there exists 2^n distinct values c and some c dependent u_c such that $\mathcal{L}_g[u_c \oplus s, c] = 0$ for all s in $\mathcal{S}$. Such a vector space exists because the row indices $(\ell||r)$ for r in $R = \{(0||r) \mid r \in \mathbb{F}_2^n\}$ and a fixed ℓ of each vertical segment in the LAT of f becomes an affine space $\mu^t(\ell||0) \oplus \mu^t(R)$ in the LAT of g, so that the image of the row indices of each of the 2^n vertical segments has an identical linear part. We thus assume that $\mathcal{S} = \mu^t(R)$.

Then, we choose an arbitrary bijective linear mapping[12] $\overline{\mu}^t$ such that $\overline{\mu}^t(\mathcal{S}) = R$. Let a^t, b^t, c^t, d^t be $n \times n$-bit matrices such that

$$\mu'^t = \overline{\mu}^t \circ \mu^t = \begin{pmatrix} a^t & c^t \\ b^t & d^t \end{pmatrix}.$$

Note that $\mu'^t(R) = \overline{\mu}^t(\mu^t(R)) = \overline{\mu}^t(\mathcal{S}) = R$, which implies $c^t = 0$.

We then apply $\overline{\mu}^t$ to columns of $\mathcal{L}_g$ to obtain a new LAT $\mathcal{L}'_g$ such that $\mathcal{L}'_g[\overline{\mu}^t(u), v] = \mathcal{L}_g[u, v]$. Using Theorem 1, we define $g' = g \circ \overline{\mu}$ so that the LAT of g' is $\mathcal{L}'_g$. Note that g' can also be expressed using f and μ':

$$g' = \eta \circ f \circ \mu \circ \overline{\mu} = \eta \circ f \circ \mu', \quad \text{with } \mu' = \begin{pmatrix} a & b \\ 0 & d \end{pmatrix}.$$

[12] We make some definitions with transpose to simplify later notations.

The function g' we obtained is such that there is no branch from the left side to the right side in the input linear layer as the corresponding element of the μ' matrix is equal to zero. We can apply the same method to the inverse of g (thus working with the transpose of the LAT) and find a linear mapping $\overline{\eta}$ allowing us to define a new permutation g'' such that:

$$g'' = \overline{\eta} \circ g \circ \overline{\mu} \text{ where } \overline{\eta} \circ \eta = \begin{pmatrix} a' & b' \\ 0 & d' \end{pmatrix}.$$

The resulting affine-equivalent structure is shown in Figure 13b. Note that the LAT of g'' exhibits the patterns described in Section 5.1. This can be explained using an alternative representation of g'' where the Feistel functions are replaced by some other affine equivalent functions as shown in Figure 13c. It also implies that we can decompose g'', as described in the next sections.

The dominating step in terms of complexity is Algorithm 1, meaning that building g'' from g takes time $\mathrm{O}(2^{6n})$ (see next section).

Subroutine: Recovering Linear Subpaces Suppose we are given the $\mathcal{L}$ of a $2n$-bit permutation. Our attack requires us to recover efficiently a linear space $\mathcal{S}$ of size 2^n such that $\mathcal{L}[s \oplus L(u), u] = 0$ for all s in $\mathcal{S}$, where u is in a linear space of size 2^n and where L is some linear permutation. Algorithm 1 is our answer to this problem.

For each column index c, we extract all s such that $|\{a, \mathcal{L}[a,c] = 0\} \cap \{a, \mathcal{L}[a, c \oplus s] = 0\}| \geq 2^n$. If s is indeed in $\mathcal{S}$ and c is a valid column index, then this intersection must contain $L(c) \oplus \mathcal{S}$, which is of size 2^n. If it is not the case, we discard s. Furthermore, if c is a valid column index, then there must be at least 2^n such s as all s in $\mathcal{S}$ have this property: this allows us to filter out columns not yielding enough possible s. For each valid column, the set of offsets s extracted must contain $\mathcal{S}$. Thus, taking the intersection of all these sets yields $\mathcal{S}$ itself.

To increase filtering, we use a simple heuristic function `refine`$(\mathcal{Z}, n)$ which returns the subset $\overline{\mathcal{Z}}$ of $\mathcal{Z}$ such that, for all z in $\overline{\mathcal{Z}}$, $|(z \oplus \mathcal{Z}) \cap \mathcal{Z}| \geq 2^n$. The key observation is that if $\mathcal{Z}$ contains a linear space of size at least 2^n then $\overline{\mathcal{Z}}$ contains it too. This subroutine runs in time $\mathrm{O}(|\mathcal{Z}|^2)$.

The dominating step in the time complexity of this algorithm is the computation of $|(s \oplus \mathcal{Z}) \cap \mathcal{Z}|$ for every c and s. The complexity of this step is $\mathrm{O}(2^{2n} \times 2^{2n} \times |\mathcal{Z}|)$. A (loose) upper bound on the time complexity of this algorithm is thus $\mathrm{O}(2^{6n})$ where n is the branch size of the inner Feistel Network, i.e. half of the block size.

Second Step: Using a Yoyo Game The decomposition of $\mathsf{AF^4A}$ has now been reduced to attacking g'', a $2n$-bit 4-round Feistel Network composed with two n-bit linear permutations A and B. The next step is to recover these linear permutations. To achieve this, we use a simple observation inspired by the so-called *yoyo game* used in [31] to attack a 5-round FN.

Algorithm 1 Linear subspace extraction
Inputs LAT $\mathcal{L}$, branch size n — **Output** Linear space $\mathcal{S}$

```
$\mathcal{C} := \emptyset$
for all $c \in [1, 2^{2n} - 1]$ do
    $\mathcal{Z} := \{i \in [1, 2^{2n} - 1], \mathcal{L}[i, c] = 0\}$
    if $\mathcal{Z} \geq 2^n$ then
        $\mathcal{S}_c := \emptyset$                                  ▷ $\mathcal{S}_c$ is the candidate for $\mathcal{S}$ for $c$.
        for all $s \in [1, 2^{2n} - 1]$ do
            if $|(s \oplus \mathcal{Z}) \cap \mathcal{Z}| \geq 2^n$ then
                $S_c := S_c \cup \{s\}$
            end if
        end for
        $\mathcal{S}_c :=$ refine$(\mathcal{S}_c, 2^n)$
        if $|\mathcal{S}_c| \geq 2^n$ then
            Store $\mathcal{S}_c$ ; $\mathcal{C} := \mathcal{C} \cup \{c\}$
        end if
    end if
end for
$\mathcal{C} :=$ refine$(\mathcal{C}, 2^n)$ ; $\mathcal{S} := [0, 2^{2n} - 1]$
for all $c \in \mathcal{C}$ do
    $\mathcal{S} := \mathcal{S} \cap \mathcal{S}_c$
end for
return $\mathcal{S}$
```

Consider the differential trail parametrized by $\gamma \neq 0$ described in Figure 14. If the pair of plaintexts $(x_L||x_R, x'_L||x'_R)$ follows the trail (i.e. is connected in γ), then so does $(x_L \oplus \gamma||x_R, x'_L \oplus \gamma||x'_R)$. Furthermore, if $(y_L||y_R) = g''(x_L||x_R)$ and $(y'_L||y'_R) = g''(x'_L||x'_R)$, then swapping the right output words and decrypting the results leads to a pair of plaintexts $\left(g''^{-1}(y_L||y'_R), g''^{-1}(y'_L||y_R)\right)$ that still follows the trail. It is thus possible to iterate the addition of γ and the swapping of the right output word to generate many right pairs. More importantly, if x and x' do follow the trail, iterating these transformation must lead to the difference on the right output word being constant and equal to $B(\gamma)$: if it is not the case, we can abort and start again from another pair x, x'.

We can thus recover B fully by trying to iterate the game described above for random pairs (x, x') and different values of γ. Once a good pair has been found, we deduce that $B(\gamma)$ is the difference in the right output words of the ciphertext pairs obtained. We thus perform this step for $\gamma = 1, 2, 4, 8$, etc., until the image by B of all bits has been found. Wrong starting pairs are identified quickly so this step takes time $\mathrm{O}(n2^{2n})$. Indeed, we need to recover n linearly independent n-bit vectors; for each vector we try all 2^n candidates and for each guess we run a Yoyo game in time 2^n to check the guess. The other linear part, A, is recovered identically by running the same attack while swapping the roles of encryption and decryption.

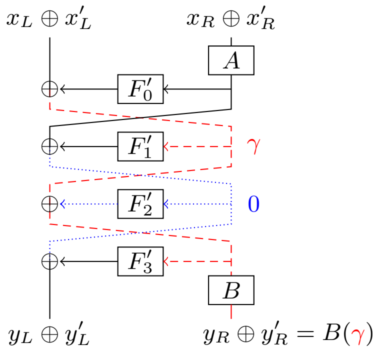


Fig. 14: The differential trail used to recover B.

Final Step: a Full Decomposition As stated before, we can recover all 4 Feistel functions in g'' minus its linear layers in time $O(2^{3n/2})$ using the guess and determine approach described in [31]. This gives us a 4-round FN, denoted $\mathcal{F}$, which we can use to decompose g like so (where I denotes the identity matrix):

$$g = \overline{\eta}^{-1} \circ \begin{pmatrix} I & 0 \\ 0 & B \end{pmatrix} \circ \mathcal{F} \circ \begin{pmatrix} I & 0 \\ 0 & A \end{pmatrix} \circ \overline{\mu}^{-1}.$$

5.3 Outline of an Attack Against $\mathsf{AF^3A}$

A structure having one less Feistel round could be attacked in a similar fashion. The main modifications would be as follows.

1. The pattern in the LAT we try to rebuild would not be the one described in Section 5.1 but a white square similar to the one observed in the LAT of π'. Indeed, an integral distinguisher similar to the one existing in π' exists for any 3-round FN when the second Feistel function is a bijection.
2. Recovering the remaining $(n \times n)$ mappings with a yoyo game would be much more efficient since there would not be any need to guess that a pair follows the trail.

The complexity of such an attack would be dominated as before by the recovery of the linear subspace embedded in the LAT so that it would take time $O(2^{6n})$.

5.4 Comments on Affine-Whitened Feistel Networks

Table 9 contains a comparison of the complexities of the attack recovering the Feistel functions of Feistel Networks along with, possibly, the linear layers used to whiten it.

The complexities of our attacks against $\mathsf{AF^kA}$ are dominated by the linear subspace recovery which is much slower than an attack against as much as 5 Feistel Network rounds. It seems like using affine whitening rather than a

Target	Type	Time Complexity	Ref.
$\mathsf{AF}^3\mathsf{A}$	LAT-based	2^{6n}	Sec. 5.3
F^4	Guess & Det.	$2^{3n/2}$	[31]
$\mathsf{AF}^4\mathsf{A}$	LAT-based	2^{6n}	Sec. 5.2
F^5	Yoyo cryptanalysis	2^{2n}	[31]

Table 9: Complexity of recovery attacks against (possibly linearly whitened) Feistel Networks with n-bit branches and secret bijective Feistel functions.

simpler XOR-based whitening increases the complexity of a cryptanalysis significantly. This observation can be linked with the recent attacks against the ASASA scheme [30]: while attacking SAS is trivial, the cryptanalysis of ASASA requires sophisticated algorithms.

We note that a better algorithm for linear subspace extraction will straightforwardly lead to a lower attack complexity. However, the complexity is lower bounded by LAT computation which is $O(n2^{4n})$ in our case.

For the sake of completeness, we tried this attack against the "F-Table" of Skipjack [9]. It failed, meaning that it has neither an $\mathsf{AF}^3\mathsf{A}$ nor an $\mathsf{AF}^4\mathsf{A}$ structure. Note also that, due to the presence of the white square in the LAT of $\hat{\pi}$, running the linear subspace recovery described in Algorithm 1 on π returns the vector space $\mathcal{V}$ which allowed to start our decomposition of this S-Box.

We implemented the first step of our attack (including Algorithm 1) using SAGE [35] and ran it on a computer with 16 Intel Xeon CPU (E5-2637) v3 clocked at 3.50 GHz. It recovers correct linear layers $\overline{\eta}$ and $\overline{\mu}$ in about 3 seconds for $n = 4$, 14 seconds for $n = 5$, 4 minutes for $n = 6$ and 1 hour for $n = 7$. Since the first step is the costliest, we expect a complete attack to take a similar time.

6 Conclusion

The S-Box used by the standard hash function Streebog, the standard block cipher Kuznyechik and the CAESAR first round candidate STRIBOBr1 has a hidden structure. Using the three non-linear 4-bit permutations ν_0, ν_1 and σ, the non-linear 4-bit function ϕ and the 8-bit linear permutations α and ω, the computation of $\pi(\ell||r)$ can be made as follows:

1. $(\ell||r) := \alpha(\ell||r)$
2. If $r = 0$ then $\ell := \nu_0(\ell)$, else $\ell := \nu_1(\ell \odot r^{14})$
3. $r := \sigma\big(r \odot \phi(l)\big)$
4. Return $\omega(\ell||r)$

How and why those components were chosen remains an open question. Indeed, their individual cryptographic properties are at best not impressive and, at worst, downright bad. However, knowing this decomposition allows a much more efficient hardware implementation of the S-Box.

We also described a decomposition attack against $\mathsf{AF}^4\mathsf{A}$ which uses the same high level principles as our attack against π: first spot patterns in the LAT, then deduce the whitening linear layers and finally break the core.

7 Acknowledgment

We thank Yann Le Corre for studying the hardware implementation of the S-Box. We also thank Oleksandr Kazymyrov for suggesting this target and the anonymous reviewers for their helpful comments. The work of Léo Perrin is supported by the CORE ACRYPT project (ID C12-15-4009992) funded by the *Fonds National de la Recherche* (Luxembourg). The work of Aleksei Udovenko is supported by the *Fonds National de la Recherche*, Luxembourg (project reference 9037104).

References

1. Daemen, J., Rijmen, V.: The design of Rijndael: AES-the advanced encryption standard. Springer (2002)
2. Biham, E., Shamir, A.: Differential cryptanalysis of DES-like cryptosystems. Journal of CRYPTOLOGY **4**(1) (1991) 3–72
3. Tardy-Corfdir, A., Gilbert, H.: A known plaintext attack of FEAL-4 and FEAL-6. In Feigenbaum, J., ed.: Advances in Cryptology – CRYPTO'91. Volume 576 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (1992) 172–182
4. Matsui, M.: Linear cryptanalysis method for DES cipher. In: Advances in Cryptology – EUROCRYPT'93, Springer (1994) 386–397
5. Nyberg, K.: Differentially uniform mappings for cryptography. In: Advances in cryptology — Eurocrypt'93, Springer (1994) 55–64
6. Suzaki, T., Minematsu, K., Morioka, S., Kobayashi, E.: TWINE: A Lightweight Block Cipher for Multiple Platforms. In: Selected Areas in Cryptography, Springer (2013) 339–354
7. U.S. DEPARTMENT OF COMMERCE/National Institute of Standards and Technology: Data encryption standard. Federal Information Processing Standards Publication (1999)
8. Coppersmith, D.: The Data Encryption Standard (DES) and its strength against attacks. IBM journal of research and development **38**(3) (1994) 243–250
9. National Security Agency, N.S.A.: SKIPJACK and KEA Algorithm Specifications (1998)
10. Biryukov, A., Perrin, L.: On Reverse-Engineering S-Boxes with Hidden Design Criteria or Structure. In: Advances in Cryptology – CRYPTO 2015. Lecture Notes in Computer Science. Springer Berlin Heidelberg (2015) (to appear)
11. Federal Agency on Technical Regulation and Metrology: GOST R 34.11-2012: Streebog hash function (2012) `https://www.streebog.net/`.
12. Guo, J., Jean, J., Leurent, G., Peyrin, T., Wang, L.: The Usage of Counter Revisited: Second-Preimage Attack on New Russian Standardized Hash Function. In Joux, A., Youssef, A., eds.: Selected Areas in Cryptography – SAC 2014. Volume 8781 of Lecture Notes in Computer Science. Springer International Publishing (2014) 195–211

13. AlTawy, R., Youssef, A.M.: Watch your constants: Malicious Streebog. IET Information Security (2015)
14. Rudskoy, V.: Note on Streebog constants origin (2015) http://www.tc26.ru/en/ISO_IEC/streebog/streebog_constants_eng.pdf.
15. Shishkin, V., Dygin, D., Lavrikov, I., Marshalko, G., Rudskoy, V., Trifonov, D.: Low-weight and hi-end: Draft Russian Encryption Standard. CTCrypt'14, 05-06 June 2014, Moscow, Russia. Preproceedings (2014) 183–188
16. Federal Agency on Technical Regulation and Metrology: Block ciphers (2015) http://www.tc26.ru/en/standard/draft/ENG_GOST_R_bsh.pdf.
17. AlTawy, R., Youssef, A.M.: A meet in the middle attack on reduced round Kuznyechik. Cryptology ePrint Archive, Report 2015/096 (2015) http://eprint.iacr.org/.
18. Dolmatov, V.: GOST 28147-89: Encryption, decryption, and message authentication code (MAC) algorithms. http://www.rfc-editor.org/rfc/rfc5830.txt (March 2010) RFC 5830.
19. Saarinen, M.J.O.: STRIBOB: Authenticated encryption from GOST R 34.11-2012 LPS permutation. IACR Cryptology ePrint Archive **2014** (2014) 271
20. Barreto, P., Rijmen, V.: The Whirlpool hashing function. In: First open NESSIE Workshop, Leuven, Belgium. Volume 13. (2000) 14
21. Saarinen, M.J.O.: STRIBOBr2 availability. Mail to the CAESAR mailing list (https://groups.google.com/forum/#!topic/crypto-competitions/_zgi54-NEFM)
22. Shishkin, V.: Принципы синтеза перспективного алгоритма блочного шифрования с длиной блока 128 бит (2013) http://www.ruscrypto.ru/resource/summary/rc2013/ruscrypto_2013_066.zip.
23. Saarinen, M.J.O., Brumley, B.B.: WhirlBob, the Whirlpool variant of STRIBOB. Cryptology ePrint Archive, Report 2014/501 (2014) http://eprint.iacr.org/.
24. Knudsen, L.R., Robshaw, M.J., Wagner, D.: Truncated differentials and Skipjack. In: Advances in Cryptology–CRYPTO'99, Springer (1999) 165–180
25. Biham, E., Biryukov, A., Dunkelman, O., Richardson, E., Shamir, A.: Initial observations on Skipjack: Cryptanalysis of Skipjack-3xor. In: Selected Areas in Cryptography, Springer (1999) 362–375
26. Knudsen, L., Wagner, D.: On the structure of Skipjack. Discrete Applied Mathematics **111**(1) (2001) 103–116
27. Kazymyrov, O., Kazymyrova, V.: Algebraic Aspects of the Russian Hash Standard GOST R 34.11-2012. IACR Cryptology ePrint Archive **2013** (2013) 556
28. Biryukov, A., Shamir, A.: Structural cryptanalysis of SASAS. In Pfitzmann, B., ed.: Advances in Cryptology – EUROCRYPT 2001. Volume 2045 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2001) 395–405
29. Dinur, I., Dunkelman, O., Kranz, T., Leander, G.: Decomposing the ASASA block cipher construction. Cryptology ePrint Archive, Report 2015/507 (2015) http://eprint.iacr.org/.
30. Minaud, B., Derbez, P., Fouque, P.A., Karpman, P.: Key-recovery attacks on ASASA. In Iwata, T., Cheon, J.H., eds.: Advances in Cryptology - ASIACRYPT 2015. Volume 8270 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2015) To appear
31. Biryukov, A., Leurent, G., Perrin, L.: Cryptanalysis of Feistel networks with secret round functions. In Dunkelman, O., Keliher, L., eds.: Selected Areas in Cryptography – SAC 2015. Lecture Notes in Computer Science. Springer International Publishing (2015) To appear

32. Daemen, J., Rijmen, V.: Probability distributions of correlation and differentials in block ciphers. Journal of Mathematical Cryptology JMC **1**(3) (2007) 221–242
33. Blondeau, C., Canteaut, A., Charpin, P.: Differential properties of power functions. International Journal of Information and Coding Theory **1**(2) (2010) 149–170
34. Preneel, B.: Analysis and design of cryptographic hash functions. PhD thesis, PhD thesis, Katholieke Universiteit Leuven (1993)
35. The Sage Developers: Sage Mathematics Software (Version 6.8). (2015) `http://www.sagemath.org`.
36. Canright, D.: A very compact S-Box for AES. In Rao, J., Sunar, B., eds.: Cryptographic Hardware and Embedded Systems – CHES 2005. Volume 3659 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2005) 441–455
37. Bogdanov, A., Leander, G., Nyberg, K., Wang, M.: Integral and Multidimensional Linear Distinguishers with Correlation Zero. In Wang, X., Sako, K., eds.: Advances in Cryptology – ASIACRYPT 2012. Volume 7658 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2012) 244–261
38. Gérard, B., Grosso, V., Naya-Plasencia, M., Standaert, F.X.: Block ciphers that are easier to mask: how far can we go? In: Cryptographic Hardware and Embedded Systems-CHES 2013. Springer (2013) 383–399
39. Standaert, F.X., Piret, G., Rouvroy, G., Quisquater, J.J., Legat, J.D.: ICEBERG : An Involutional Cipher Efficient for Block Encryption in Reconfigurable Hardware. In Roy, B., Meier, W., eds.: Fast Software Encryption. Volume 3017 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2004) 279–298
40. Barreto, P., Rijmen, V.: The Khazad legacy-level block cipher. Primitive submitted to NESSIE **97** (2000)
41. Shirai, T., Shibutani, K., Akishita, T., Moriai, S., Iwata, T.: The 128-bit blockcipher CLEFIA. In: Fast software encryption, Springer (2007) 181–195
42. Grosso, V., Leurent, G., Standaert, F.X., Varıcı, K.: LS-designs: Bitslice encryption for efficient masked software implementations. In: Fast Software Encryption. (2014)
43. Canteaut, A., Duval, S., Leurent, G.: Construction of Lightweight S-Boxes using Feistel and MISTY structures (Full Version). Cryptology ePrint Archive, Report 2015/711 (2015) `http://eprint.iacr.org/`.

A On the Choice of the Round Constants

The round constants of Streebog were chosen by feeding 12 different seeds into a round-constant-less version of the hash function with a modified linear layer [14]. These seeds are given as hexadecimal strings of varying length which seem at first glance to lack any justification. However, they correspond to Russian names written backwards in cyrillic and encoded in `cp1251` as described in Table 10.

B Definition of the S-Box

Table 11 contains the definition of π.

C The Mini-Block Ciphers

Tables 12a and 12b contain the mini-block ciphers T and U respectively.

R	Hexadecimal seed	Name
1	`e2e5ede1e5f0c3`	Гребнев (Grebnev)
2	`f7e8e2eef0e8ece8e4e0ebc220e9e5e3f0e5d1`	Сергей Владимирович (Sergej Vladimirovich)
3	`f5f3ecc4`	Дмух (Dmukh)
4	`f7e8e2eef0e4ede0f1eae5ebc020e9e5f0e4edc0`	Андрей Александрович (Andrej Aleksandrovich)
5	`ede8e3fbc4`	Дыгин (Dygin)
6	`f7e8e2eeebe9e0f5e8cc20f1e8ede5c4`	Денис Михайлович (Denis Mihajlovich)
7	`ede8f5fef2e0cc`	Матюхин (Matjuhin)
8	`f7e8e2eef0eef2eae8c220e9e8f0f2e8ecc4`	Дмитрий Викторович (Dmitrij Viktorovich)
9	`e9eeeaf1e4f3d0`	Рудской (Rudskoj)
10	`f7e8e2e5f0eee3c820f0e8ece8e4e0ebc2`	Владимир Игоревич (Vladimir Igorevich)
11	`ede8eaf8e8d8`	Шишкин (Shishkin)
12	`f7e8e2e5e5f1eae5ebc020e9e8ebe8f1e0c2`	Василий Алексеевич (Vasilij Alekseevich)

Table 10: The seeds used for each round constant and the corresponding name.

	.0	.1	.2	.3	.4	.5	.6	.7	.8	.9	.a	.b	.c	.d	.e	.f
0.	fc	ee	dd	11	cf	6e	31	16	fb	c4	fa	da	23	c5	04	4d
1.	e9	77	f0	db	93	2e	99	ba	17	36	f1	bb	14	cd	5f	c1
2.	f9	18	65	5a	e2	5c	ef	21	81	1c	3c	42	8b	01	8e	4f
3.	05	84	02	ae	e3	6a	8f	a0	06	0b	ed	98	7f	d4	d3	1f
4.	eb	34	2c	51	ea	c8	48	ab	f2	2a	68	a2	fd	3a	ce	cc
5.	b5	70	0e	56	08	0c	76	12	bf	72	13	47	9c	b7	5d	87
6.	15	a1	96	29	10	7b	9a	c7	f3	91	78	6f	9d	9e	b2	b1
7.	32	75	19	3d	ff	35	8a	7e	6d	54	c6	80	c3	bd	0d	57
8.	df	f5	24	a9	3e	a8	43	c9	d7	79	d6	f6	7c	22	b9	03
9.	e0	0f	ec	de	7a	94	b0	bc	dc	e8	28	50	4e	33	0a	4a
a.	a7	97	60	73	1e	00	62	44	1a	b8	38	82	64	9f	26	41
b.	ad	45	46	92	27	5e	55	2f	8c	a3	a5	7d	69	d5	95	3b
c.	07	58	b3	40	86	ac	1d	f7	30	37	6b	e4	88	d9	e7	89
d.	e1	1b	83	49	4c	3f	f8	fe	8d	53	aa	90	ca	d8	85	61
e.	20	71	67	a4	2d	2b	09	5b	cb	9b	25	d0	be	e5	6c	52
f.	59	a6	74	d2	e6	f4	b4	c0	d1	66	af	c2	39	4b	63	b6

Table 11: The S-Box in hexadecimal. For example, $\pi($`0x7a`$) =$ `0xc6`.

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
T_0	e	f	2	5	7	b	8	1	3	c	d	a	0	9	4	6
T_1	2	9	a	4	e	6	7	b	1	8	3	d	0	c	f	5
T_2	e	f	2	5	7	b	8	1	3	c	d	a	0	9	4	6
T_3	5	d	4	2	6	7	b	8	c	1	9	f	0	3	a	e
T_4	5	e	6	7	4	3	f	a	0	1	d	2	8	b	c	9
T_5	9	d	f	a	c	6	8	1	0	5	b	3	2	4	e	7
T_6	3	9	d	f	1	e	b	8	0	2	7	c	4	a	5	6
T_7	5	e	6	7	4	3	f	a	0	1	d	2	8	b	c	9
T_8	7	b	8	5	9	d	c	3	2	e	a	f	6	1	0	4
T_9	d	f	a	c	e	6	2	5	1	3	b	7	9	4	0	8
T_a	e	6	7	4	c	3	8	1	a	2	d	9	5	b	0	f
T_b	4	2	5	d	b	8	6	7	9	f	c	1	a	e	0	3
T_c	2	5	a	4	3	9	d	8	c	f	0	7	b	1	6	e
T_d	e	6	2	5	d	f	a	c	9	4	0	8	1	3	b	7
T_e	9	d	c	3	7	b	8	5	6	1	0	4	2	e	a	f
T_f	8	1	7	b	2	5	e	f	4	6	0	9	d	a	3	c

(a) T.

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
U_0	8	f	0	2	d	5	6	9	e	3	1	7	c	b	4	a
U_1	8	c	7	3	d	f	2	0	e	4	1	b	6	5	9	a
U_2	3	4	e	9	d	8	0	5	1	2	c	f	7	b	a	6
U_3	b	8	9	a	0	7	2	5	f	6	d	4	1	e	3	c
U_4	c	2	5	b	e	8	7	1	4	f	d	6	9	3	0	a
U_5	4	e	2	8	3	7	5	1	a	b	c	d	f	6	9	0
U_6	f	6	b	2	3	0	7	4	5	d	1	9	e	8	a	c
U_7	7	a	c	1	e	f	5	4	b	9	0	2	8	d	3	6
U_8	a	f	b	e	c	4	d	5	7	0	6	1	8	3	9	2
U_9	2	3	c	d	1	b	f	5	9	4	7	a	e	6	0	8
U_a	9	b	5	7	1	c	d	0	6	2	a	e	f	8	3	4
U_b	1	7	2	4	c	3	f	0	8	6	b	5	9	d	a	e
U_c	6	d	e	5	2	c	a	4	3	f	b	7	1	0	9	8
U_d	e	1	9	6	f	3	8	4	d	b	a	c	7	5	0	2
U_e	5	9	0	c	f	4	a	1	2	d	7	8	6	b	3	e
U_f	d	5	7	f	2	b	8	1	c	9	6	3	0	e	a	4

(b) U.

Table 12: The mini-block ciphers used to decompose π'^{-1}.

D Proof of the Value of the Biases in the "Stripe"

Biases in the stripe correspond to approximations $(a_L||a_R \rightsquigarrow b_L||0)$ in the linear layer-less version of π, which is denoted $\hat{\pi}$. These approximations are equal to:

$$2 \times \mathcal{L}_{a_L||a_R, b_L||0} = \sum_{x=0}^{255} (-1)^{(a_L||a_R)\cdot x \,\oplus\, (b_L||0)\cdot\hat{\pi}(x)},$$

which we decompose by splitting $x \in \mathbb{F}_2^8$ into $(l, r) \in (\mathbb{F}_2^4)^2$ to obtain

$$\sum_{r=1}^{16}\sum_{l=0}^{16} (-1)^{a_L\cdot l \,\oplus\, a_R\cdot r \,\oplus\, b_l\cdot\nu_1(l\odot\mathcal{I}(r))} + \sum_{l=0}^{16} (-1)^{a_L\cdot l \,\oplus\, b_L\cdot\nu_0(l)}. \tag{5}$$

The second term in this sum is equal to $2 \times \mathcal{L}_0[a_L, b_L]$ where $\mathcal{L}_0$ is the LAT of ν_0. The first term can be simplified using the change of variable $x = \nu_1(l \odot \mathcal{I}(r))$, i.e. $l = \nu_1^{-1}(x) \odot r$:

$$\sum_{l=0}^{16} (-1)^{a_L\cdot l \,\oplus\, b_L\cdot\nu_1(l\odot\mathcal{I}(R))} = \sum_{x=0}^{16} (-1)^{b_L\cdot x \,\oplus\, a_L\cdot(\nu_1^{-1}(x)\odot r)}.$$

As a consequence, the first term of Equation (5) can be re-written:

$$\begin{aligned}&\sum_{r=1}^{16}\sum_{x=0}^{16} (-1)^{a_R\cdot r \,\oplus\, b_L\cdot x \,\oplus\, a_L\cdot(\nu_1^{-1}(x)\odot r)} \\ &= \sum_{x=0}^{16} (-1)^{b_L\cdot x} \left(\sum_{r=0}^{16} (-1)^{a_R\cdot r \,\oplus\, a_L\cdot(\nu_1^{-1}(x)\odot r)} - 1 \right).\end{aligned}$$

First, we note that $\sum_{x=0}^{16}(-1)^{b_L \cdot x}$ is equal to 0 if $b_L \neq 0$ and 16 otherwise. Then, we remark that $\sum_{r=0}^{16}(-1)^{a_R \cdot r \,\oplus\, a_L \cdot (\nu_1^{-1}(x) \odot r)}$ is equal to $2 \times \mathcal{L}^m_{\nu_1^{-1}(x)}[a_R, a_L]$, where $\mathcal{L}^m_k$ is the LAT of the Boolean linear permutation $r \mapsto r \odot k$. If we further replace x by $y = \nu_1^{-1}(x)$ then Equation (5) can be re-written

$$2 \times \sum_{y=0}^{16}(-1)^{b_L \cdot \nu_1(y)} \mathcal{L}^m_y[a_R, a_L] - 16 \times \hat{\delta}(b_L).$$

where $\hat{\delta}(b_L) = 1$ if and only if $b_L = 0$. Besides, by experimentally exhausting all cases, we can check that the following property holds.

Remark 3. For every pair (a_R, a_L) with $a_R > 0$ and $a_L > 0$, there is exactly one value y_0 such that $\mathcal{L}^m_{y_0}[a_R, a_L] = 8$ and $\mathcal{L}^m_y[a_R, a_L] = 0$ for $y \neq y_0$.

We deduce that the expression of $\mathcal{L}[a_L||a_R, b_L||0]$ is

$$\mathcal{L}[a_L||a_R, b_L||0] = \mathcal{L}_0[a_L, b_L] + 8 \times \big((-1)^{b_L \cdot y_0} - \hat{\delta}(b_L)\big).$$

E Generating π from our Decomposition

The following SAGE [35] script generates and prints π. It can be downloaded on github: https://github.com/picarresursix/GOST-pi.

```
from sage.all import *

X = GF(2).polynomial_ring().gen()
F = GF(2**4, name="a", modulus=X**4+X**3+1)

inv   = [0x0,0x1,0xc,0x8,0x6,0xf,0x4,0xe,0x3,0xd,0xb,0xa,0x2,0x9,0x7,0x5]
nu_0  = [0x2,0x5,0x3,0xb,0x6,0x9,0xe,0xa,0x0,0x4,0xf,0x1,0x8,0xd,0xc,0x7]
nu_1  = [0x7,0x6,0xc,0x9,0x0,0xf,0x8,0x1,0x4,0x5,0xb,0xe,0xd,0x2,0x3,0xa]
sigma = [0xc,0xd,0x0,0x4,0x8,0xb,0xa,0xe,0x3,0x9,0x5,0x2,0xf,0x1,0x6,0x7]
phi   = [0xb,0x2,0xb,0x8,0xc,0x4,0x1,0xc,0x6,0x3,0x5,0x8,0xe,0x3,0x6,0xb]

alpha = Matrix(GF(2), 8, 8, [
    0,0,0,0,1,0,0,0, 0,1,0,0,0,0,0,1,
    0,1,0,0,0,0,1,1, 1,1,1,0,1,1,1,1,
    1,0,0,0,1,0,1,0, 0,1,0,0,0,1,0,0,
    0,0,0,1,1,0,1,0, 0,0,1,0,0,0,0,0,
])
omega = Matrix(GF(2), 8, 8, [
    0,0,0,0,1,0,1,0, 0,0,0,0,0,1,0,0,
    0,0,1,0,0,0,0,0, 1,0,0,1,1,0,1,0,
    0,0,0,0,1,0,0,0, 0,1,0,0,0,1,0,0,
    1,0,0,0,0,0,1,0, 0,0,0,0,0,0,0,1,
])

def applymat8(x, mat):
    y = mat * Matrix(GF(2), 8, 1, map(int, bin(x)[2:].zfill(8)))
    return int("".join(map(str, y.T[0][:8])), 2)

def F_mult(x, y):
    return (F.fetch_int(x) * F.fetch_int(y)).integer_representation()

pi = []
for x in xrange(256):
    x = applymat8(x, alpha)
    l, r = x >> 4, x & 0xf
    l = (r == 0) * nu_0[l] + (r != 0) * nu_1[F_mult(l, inv[r])]
    r = sigma[F_mult(r, phi[l])]
    x = applymat8((l << 4) | r, omega)
    pi.append(x)
print pi
```