The design of Xoodoo and Xoofff

Joan Daemen[2], Seth Hoffert, Gilles Van Assche[1] and Ronny Van Keer[1]

[1] STMicroelectronics
[2] Radboud University

Abstract. This paper presents XOODOO, a 48-byte cryptographic permutation with excellent propagation properties. Its design approach is inspired by KECCAK-p, while it is dimensioned like Gimli for efficiency on low-end processors. The structure consists of three planes of 128 bits each, which interact per 3-bit columns through mixing and nonlinear operations, and which otherwise move as three independent rigid objects. We analyze its differential and linear propagation properties and, in particular, prove lower bounds on the weight of trails using the tree search-based technique of Mella et al. (ToSC 2017). XOODOO's primary target application is in the Farfalle construction that we instantiate for the doubly-extendable cryptographic keyed (or deck) function XOOFFF. Combining a relatively narrow permutation with the parallelism of Farfalle results in very efficient schemes on a wide range of platforms, from low-end devices to high-end processors with vector instructions.

Keywords: permutation-based cryptography · Farfalle · deck function · differential cryptanalysis · linear cryptanalysis

1 Introduction

Designing a symmetric cryptographic primitive involves careful trade-offs between performance and security. For the former, an interesting challenge is to yield excellent performance on a wide range of targets, in particular from the low-end processors as used for embedded devices to the high-end server processors. This is especially useful if such a mixture of devices have to interact together. For the latter, while the security of a primitive cannot be measured and relies on public scrutiny by skilled cryptanalysts, a good design typically starts with a round function that mixes bits and frustrates the propagation of differences and linear correlations as quickly as possible.

A way to achieve performance on a wide range of targets consists in combining a high level of parallelism with a relatively small building block. On a low-end target, the computation is done serially with a small footprint, and the high-end processor can fully exploit its capabilities with the evaluation of multiple instances of the building block in parallel. As a concrete example, the Farfalle construction is a mode of use based on a permutation that allows a very high level of parallelism [BDH+17]. Instantiated with a permutation of relatively small size, it offers thus such a potential.

In [BDH+17], the authors define Kravatte by instantiating Farfalle with the 200-byte wide permutation KECCAK-p[1600] with 6 rounds. In general, Kravatte is very fast on a wide range of platforms, but there are some exceptions due to the large width of the permutation. First, on low-end processors the computation is slowed down by swapping the permutation state in and out of registers. This effect plays independently of the input and output lengths. Second, Kravatte with short input and output has a relatively large overhead per byte, and the cost of computing, say, a MAC over a message of 20 bytes is the same as for a MAC over a 199-byte message. It therefore makes sense to consider instantiating Farfalle with a narrower permutation.

Licensed under Creative Commons License CC-BY 4.0.
IACR Transactions on Symmetric Cryptology ISSN 2519-173X, Vol. 2018, No. 4, pp. 1–38
DOI:10.13154/tosc.v2018.i4.1-38

The width of the permutation, denoted b, is lower bounded by generic attacks on the Farfalle construction. Exhaustive key search on any mask can be done with a computational complexity of 2^b Farfalle executions, and generating inputs that collide in the internal state can be done with data complexity $2^{b/2}$ Farfalle executions. If the target security strength is an overall 128-bit level, both in data and computation, this lower bounds b to 256 bits. Actually taking $b = 256$ then implies a *hermetic* approach: the differential properties of the permutation used in the compression phase should not allow collision attacks better than generic ones. This condition can be relaxed by increasing b.

It therefore makes sense to consider instantiating Farfalle with a narrow permutation, yet larger than 256 bits. The Keccak-p[800] permutation is well-suited for 32-bit processors, but its width of 100 bytes makes it still rather large. Next, taking Keccak-p[400] is problematic as it is defined in terms of operations on 16-bit *lanes*. Alternatively, the Gimli permutation [BKL+17] has the interesting and inspiring feature that its state of 384 bits and its round function lend themselves nicely to low-end 32-bit processors, but also vectorization and dedicated hardware. Unfortunately, its propagation properties are less than what could be expected. For constructing a Farfalle instance with 128-bit security, one would have to take a relatively high number of rounds. We therefore took the initiative to design a permutation with the same width and objectives as Gimli, but with more favorable propagation properties. We called the result Xoodoo, and it can be seen as a porting of the Keccak-p design approach to a Gimli-shaped state.

1.1 Farfalle and deck functions

Although Xoodoo could be used in other permutation-based modes, in this paper we focus on the Farfalle construction as its primary target application of. This construction instantiates a pseudorandom function (PRF) and allows building all types of (authenticated) encryption and MAC functions, while exploiting arbitrary degrees of parallelism [BDH+17].

A Farfalle instance takes a sequence of strings as input and outputs an arbitrary number of bits. We denote a sequence of m strings $X^{(0)}$ to $X^{(m-1)}$ as $X^{(m-1)} \circ \cdots \circ X^{(1)} \circ X^{(0)}$. Its output is extendable, since the caller can request for more output bits at a low incremental cost. Similarly, the input enjoys a specific extension property: computing $F(Y \circ X)$ costs only the processing of Y if $F(X)$ was previously computed.

Clearly, Farfalle is not the only way to build functions with such properties. In order to decouple the external behavior of the function (PRF with extension properties) from an implementation (here, Farfalle), we propose to call them *Doubly-Extendable Cryptographic Keyed functions* or *deck functions* for short.

Definition 1. A *deck function* takes as input a secret key K and a sequence of an arbitrary number of strings $X^{(m-1)} \circ \cdots \circ X^{(0)} \in (\mathbb{Z}_2^*)^+$, produces a potentially infinite string of bits and takes from it the range starting from a specified offset $q \in \mathbb{N}$ and for a specified length $n \in \mathbb{N}$. We denote this as

$$Z = 0^n + F_K\left(X^{(m-1)} \circ \cdots \circ X^{(0)}\right) \ll q \ .$$

A deck function should allow efficient incremental computing. In particular, by keeping state after computing an output for input sequence $X = X^{(m-1)} \circ \cdots \circ X^{(0)}$, computing an output for $Y^{(n-1)} \circ \cdots \circ Y^{(0)} \circ X$ should have a cost independent of X.

1.2 Contributions

Our main contributions are (1) the permutation Xoodoo, with strong bounds on trail weights, and (2) the deck function Xoofff, an instance of Farfalle on top of it. Furthermore, we provide (3) an improved attack strategy for generating accumulator collisions in Farfalle.

Xoodoo is an iterated permutation that is inspired by Keccak-p and Gimli [BKL+17, BDPA11b], with a novel structure consisting of three planes of 4×32 bits each. The three planes interact per 3-bit columns through a column parity mixer [SD18] and a degree-2 nonlinear operation, while they move as three independent rigid objects for dispersion.

We show that Xoodoo scores very well with respect to avalanche metrics and has excellent differential propagation and correlation properties. In particular, we prove lower bounds on the weight of trails using the tree search-based technique of Mella et al. [MDA17], although using finer-grained units than what was proposed for Keccak-p. The increased symmetry, the two dispersion layers and the involutive nature of the nonlinear layer make Xoodoo easier to analyze than Keccak-p. In particular, it allows us obtain better trail bounds.

Finally, this paper analyzes Xoofff, its rolling functions and the properties of Xoodoo in the light of what is needed for Xoofff to be secure. From a user perspective, Xoofff is an efficient deck function that can be used for building stream ciphers, MAC functions and full-featured authenticated encryption schemes, as proposed in [BDH+17]. We provide benchmarks on some low-end and high-end processors.

1.3 Design philosophy

In this section, we discuss the design philosophy underlying Xoofff, in the more general context of building cryptographic schemes in a modular way.

Typically, one specifies a cryptographic scheme as a mode on top of a primitive, where one can prove the scheme secure on the condition that the primitive is secure (or ideal, or random, ...). We define a primitive as a cryptographic object that cannot be proved secure, but rather one that has the *objective* of being secure. This objective is expressed as a security claim, and this claim can be used by cryptanalysts to challenge the primitive. In the security proof of the mode, the statements in the security claim are assumed to be true, and so the security of the scheme is conditional on the validity of these statements.

In the case of this paper, what is the mode that is provably secure assuming the primitive is secure, and what is the primitive that must be cryptanalyzed? Here, the primitive is Xoofff and provably secure modes are modes on top of Xoofff, such as those defined in [DHAK18a].

Xoofff is a primitive but, like many other primitives, it has been built in a modular way. It makes use of the Farfalle construction and uses as building blocks the Xoodoo permutation and two rolling functions, see Figure 1. The idea behind Farfalle is not to build a secure function assuming the underlying building blocks are secure (or ideal or random). Instead, the idea is to use building blocks we know how to design in order to build an efficient cryptographic function that is useful for encryption, authentication and authenticated encryption. Of course, the propagation and algebraic properties of the underlying components are relevant and interesting, but the object to be cryptanalyzed is Xoofff and there is no security claim on the building blocks. In particular, there is no security claim on Xoodoo.

One can compare it with the way block ciphers, MAC functions or stream ciphers have been built. Here are some examples.

- Key-alternating block ciphers repeat a round function interleaved with round key addition [DR02]. The round keys are generated in a key schedule that takes the cipher key as input. No security claims are made on the round function nor the key schedule.

- Tweakable block ciphers can be built using the tweakey framework [JNP14]. Also here, there is no security reduction to underlying parts, e.g., the so-called tweakey schedule or the round function.

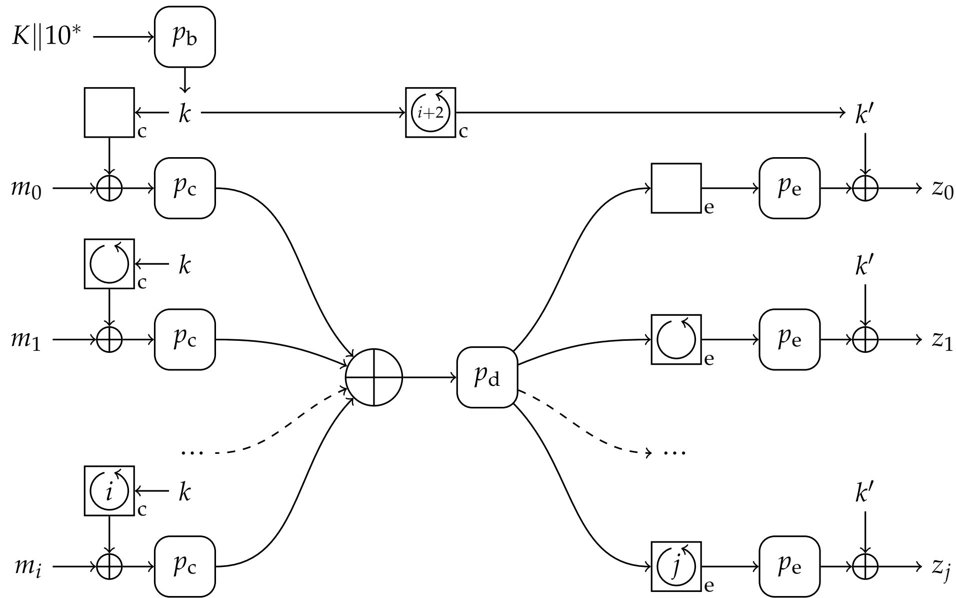

Figure 1: The Farfalle construction.

- The MAC function Pelican-MAC, an application of the Alred framework, is built from AES and a building block similar to Xoodoo: the permutation that consists of 4 unkeyed rounds of AES [DR10, DR14]. This permutation has many structural properties (e.g., symmetry, impossible differentials) not present in a random permutation, but there is no security claim on this building block, only on the function Pelican-MAC itself.
- The Salsa stream cipher can be seen as a permutation, the so-called *Salsa core*, in a certain mode [Ber08b]. The Salsa core has structural properties due to a very symmetric round function and the absence of round constants. But there is no security claim on this permutation: the construction prevents exploiting the symmetry properties, and the assumed security and target of cryptanalysis is in the Salsa stream cipher as a whole.

1.4 Outline

In Section 2, we define the Xoodoo permutation and in Section 3 the Xoofff deck function. Optimization techniques and benchmarks are presented in Section 4. The design rationale of the permutation is given in Section 5, while that of the deck function can be found in Section 6. We detail the techniques for searching differential and linear trails in Section 7. Finally, we conclude in Section 8.

2 Xoodoo specification

Xoodoo is a family of permutations parameterized by its number of rounds n_r and denoted Xoodoo$[n_\mathrm{r}]$.

Xoodoo has a classical iterated structure: It iteratively applies a round function to a state. The state consists of 3 equally sized horizontal *planes*, each one consisting of 4 parallel 32-bit *lanes*. Similarly, the state can be seen as a set of 128 *columns* of 3 bits, arranged in a 4×32 array. The planes are indexed by y, with plane $y = 0$ at the bottom

and plane $y = 2$ at the top. Within a lane, we index bits with z. The lanes within a plane are indexed by x, so the position of a lane in the state is determined by the two coordinates (x, y). The bits of the state are indexed by (x, y, z) and the columns by (x, z). *Sheets* are the arrays of three lanes on top of each other and they are indexed by x. The XOODOO state is illustrated in Figure 2.

The permutation consists of the iteration of a round function $\mathrm{R_i}$ that has 5 steps: a mixing layer θ, a plane shifting ρ_{west}, the addition of round constants ι, a non-linear layer χ and another plane shifting ρ_{east}.

We specify XOODOO in Algorithm 1, completely in terms of operations on planes and use thereby the notational conventions we specify in Table 1. We illustrate the step mappings in a series of figures: the χ operation in Figure 3, the θ operation in Figure 4, the ρ_{east} and ρ_{west} operations in Figure 5.

The round constants C_i are planes with a single non-zero lane at $x = 0$, denoted as c_i. We specify the value of this lane for indices -11 to 0 in Table 2 and refer to Appendix A for the specification of the round constants for any index.

Finally, in many applications the state must be specified as a 384-bit string s with the bits indexed by i. The mapping from the three-dimensional indexing (x, y, z) and i is given by $i = z + 32(x + 4y)$.

3 Xoofff specification and security claim

XOOFFF is a deck function obtained by applying the Farfalle construction on XOODOO[6] and two rolling functions: $\mathrm{roll_{Xc}}$ for rolling the input masks and $\mathrm{roll_{Xe}}$ for rolling the state. We specify them with operations on the lanes of the state, following the conventions of Table 1 and Table 3.

The input mask rolling function $\mathrm{roll_{Xc}}$ updates a state A in the following way:

$$\begin{aligned} A_{0,0} &\leftarrow A_{0,0} + (A_{0,0} \ll 13) + (A_{1,0} \lll 3) \\ B &\leftarrow A_0 \lll (3,0) \\ A_0 &\leftarrow A_1 \\ A_1 &\leftarrow A_2 \\ A_2 &\leftarrow B \end{aligned}$$

The state rolling function $\mathrm{roll_{Xe}}$ updates a state A in the following way:

$$\begin{aligned} A_{0,0} &\leftarrow A_{1,0} \cdot A_{2,0} + (A_{0,0} \lll 5) + (A_{1,0} \lll 13) + \texttt{0x00000007} \\ B &\leftarrow A_0 \lll (3,0) \\ A_0 &\leftarrow A_1 \\ A_1 &\leftarrow A_2 \\ A_2 &\leftarrow B \end{aligned}$$

Definition 2 (XOOFFF). XOOFFF is $\mathrm{Farfalle}[p_{\mathrm{b}}, p_{\mathrm{c}}, p_{\mathrm{d}}, p_{\mathrm{e}}, \mathrm{roll_c}, \mathrm{roll_e}]$ with the following parameters:

- $p_{\mathrm{b}} = p_{\mathrm{c}} = p_{\mathrm{d}} = p_{\mathrm{e}} = $ XOODOO[6],
- $\mathrm{roll_c} = \mathrm{roll_{Xc}}$ and
- $\mathrm{roll_e} = \mathrm{roll_{Xe}}$.

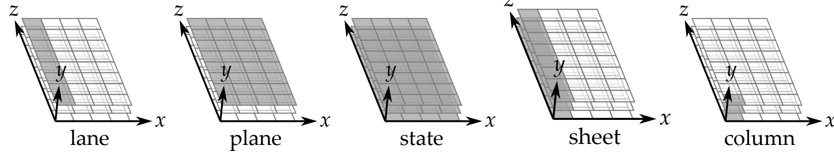

Figure 2: Toy version of the XOODOO state, with lanes reduced to 8 bits, and different parts of the state highlighted.

Table 1: Notational conventions

A_y	Plane y of state A
$A_y \lll (t, v)$	Cyclic shift of A_y moving bit in (x, z) to position $(x + t, z + v)$
$\overline{A_y}$	Bitwise complement of plane A_y
$A_y + A_{y'}$	Bitwise sum (XOR) of planes A_y and $A_{y'}$
$A_y \cdot A_{y'}$	Bitwise product (AND) of planes A_y and $A_{y'}$

Algorithm 1 Definition of XOODOO$[n_\mathrm{r}]$ with n_r the number of rounds

Parameters: Number of rounds n_r
for Round index i from $1 - n_\mathrm{r}$ to 0 **do**
$A = \mathrm{R_i}(A)$

Here $\mathrm{R_i}$ is specified by the following sequence of steps:

θ :
$$P \leftarrow A_0 + A_1 + A_2$$
$$E \leftarrow P \lll (1, 5) + P \lll (1, 14)$$
$$A_y \leftarrow A_y + E \text{ for } y \in \{0, 1, 2\}$$
ρ_west :
$$A_1 \leftarrow A_1 \lll (1, 0)$$
$$A_2 \leftarrow A_2 \lll (0, 11)$$
ι :
$$A_0 \leftarrow A_0 + \mathrm{C}_i$$
χ :
$$B_0 \leftarrow \overline{A_1} \cdot A_2$$
$$B_1 \leftarrow \overline{A_2} \cdot A_0$$
$$B_2 \leftarrow \overline{A_0} \cdot A_1$$
$$A_y \leftarrow A_y + B_y \text{ for } y \in \{0, 1, 2\}$$
ρ_east :
$$A_1 \leftarrow A_1 \lll (0, 1)$$
$$A_2 \leftarrow A_2 \lll (2, 8)$$

Table 2: The round constants c_i with $-11 \leq i \leq 0$, in hexadecimal notation (the least significant bit is at $z = 0$).

i	c_i	i	c_i	i	c_i	i	c_i
−11	0x00000058	−8	0x000000D0	−5	0x00000060	−2	0x000000F0
−10	0x00000038	−7	0x00000120	−4	0x0000002C	−1	0x000001A0
−9	0x000003C0	−6	0x00000014	−3	0x00000380	0	0x00000012

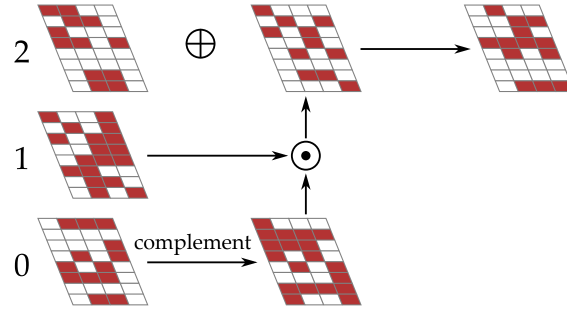

Figure 3: Effect of χ on one plane.

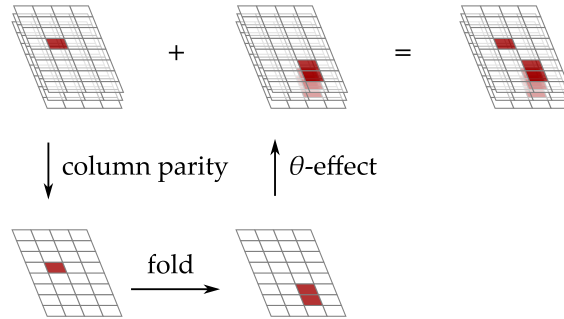

Figure 4: Effect of θ on a single-bit state.

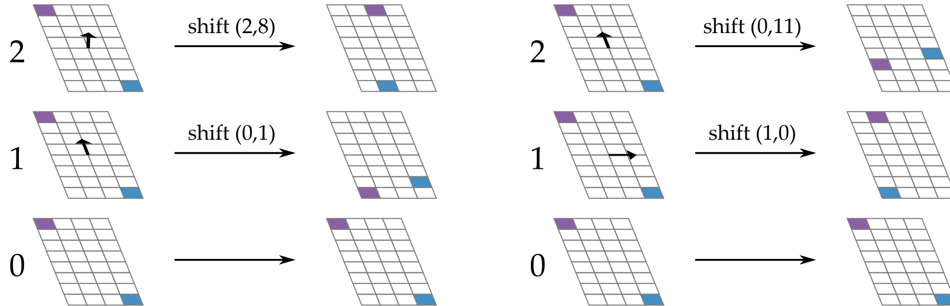

Figure 5: Illustration of ρ_{east} (left) and ρ_{west} (right).

Table 3: Notational conventions for specification of the rolling functions

$A_{y,x}$	Lane x of plane A_y
B	An auxiliary variable that has the shape of a plane
$A_{y,x} \lll v$	Cyclic shift of lane $A_{y,x}$ moving bit from x to $x+v$
$A_{y,x} \ll v$	Shift of lane $A_{y,x}$ moving bit from x to $x+v$, setting bits $x < v$ to 0
$A_{y,x} + A_{y',x'}$	Bitwise sum (XOR) of lanes $A_{y,x}$ and $A_{y',x'}$
$A_{y,x} \cdot A_{y',x'}$	Bitwise product (AND) of lanes $A_{y,x}$ and $A_{y',x'}$

We make the following security claim on Xoofff.

Claim 1. *Let* $\mathbf{K} = (K_0, \ldots, K_{u-1})$ *be an array of* u *secret keys, each uniformly and independently chosen from* $\mathbb{Z}_2^\kappa$ *with* $\kappa < 384$*. Then, the advantage of distinguishing the array of functions* $\text{Xoofff}_{K_i}(\cdot)$ *with* $i \in \mathbb{Z}_u$ *from an array of random oracles* $\mathcal{RO}(i, \cdot)$*, is at most*

$$\frac{uN + \binom{u}{2}}{2^\kappa} + \frac{N}{2^{192}} + \frac{M}{2^{128}} + \frac{\sqrt{u}N'}{2^{\kappa/2-1}} + \frac{N'}{2^{95}} . \quad (1)$$

Here,

- N *is the* computational complexity *expressed in the (computationally equivalent) number of executions of* Xoodoo[6],
- N' *is the* quantum computational complexity *expressed in the (equivalent) number of quantum oracle accesses to* Xoodoo[6]*, and*
- M *is the* online *or* data complexity *expressed in the total number of input and output blocks processed by* $\text{Xoofff}_{K_i}(\cdot)$.

In (1), the first term accounts for the effort to find one of the u secret keys by exhaustive search, and for the probability that two keys are equal. The second term expresses that the complexity of recovering the accumulator or any rolling state inside Xoofff must be as hard as recovering 192 secret bits. The third term expresses the effort to find a collision in the accumulator.

The fourth and fifth terms only apply if the adversary has access to a quantum computer. The fourth term accounts for a quantum search (or quantum amplification algorithm) to find one of the u keys [Gro96, BHMT02]. The probability of success after N' iterations is $\sin^2((2N'+1)\theta)$ with $\theta = \arcsin\sqrt{u/2^\kappa}$. We upper bound this as $2N'\sqrt{u/2^\kappa}$. The fifth term similarly accounts for a quantum search of a 192-bit secret.

Note that we assume that Xoofff is implemented on a classical computer. In other words, we do not make claims w.r.t. adversaries who would make quantum superpositions of queries to the device implementing Xoofff and holding its secret key(s).

We restrict keys to the uniform distribution to keep our claim simple and to avoid pathological cases that would not offer good security. In the multi-user setting, we require the keys to be independently drawn. If an adversary can manipulate K_i, such as in so-called *unique keys* that consist of a long-term key with a counter appended, we recommend hashing the key and the counter with a proper hash function.

We do support the use of variable-length keys in the multi-user setting, where we assume that a key of given length is selected uniformly of the strings with that length. The claimed distinguishing bound then becomes slightly more complex and is given in Equation (2):

$$\sum_{\kappa \in \mathbf{L}} \frac{u_\kappa N + \binom{u_\kappa}{2}}{2^\kappa} + \frac{N}{2^{192}} + \frac{M}{2^{128}} + \sum_{\kappa \in \mathbf{L}} \frac{\sqrt{u_\kappa}N'}{2^{\kappa/2-1}} + \frac{N'}{2^{95}} , \quad (2)$$

with $\mathbf{L}$ the array of the distinct key lengths in use and u_l the number of keys of length l.

4 Implementation aspects

In this section, we first report on some possible optimizations, then we give benchmarks for Xoofff.

4.1 Optimizations

Naturally, the lanes of Xoodoo coincide with words on 32-bit processors. All the operations in Algorithm 1 can be implemented using bitwise logical operations and rotations. The χ and θ steps can be implemented in a way that minimizes temporary storage.

The step χ is specified by a number of parallel computations but can be serialized to allow in place processing with no computational penalty. In particular, the following sequence of operations performs χ:

$$A_0 \leftarrow A_0 + \overline{A_1} \cdot A_2$$
$$A_1 \leftarrow A_1 + \overline{A_2} \cdot A_0$$
$$A_2 \leftarrow A_2 + \overline{A_0} \cdot A_1$$

For the step θ, one can exploit the fact that the θ-effect E added to a sheet x depends only on the parity of the sheet at $x - 1$. One can proceed as follows. First, one computes the lane $x = 1$ of the θ-effect E (denoted E_1) from the parity of the sheet at $x = 0$ and stores it in a temporary 32-bit register R. Then, one adds $R = E_1$ to the sheet at $x = 1$. To compute E_2 from the parity of the sheet at $x = 1$, we notice that the sheet at $x = 1$ does not have its original value anymore, but all the lanes got E_1 added to it. Hence, one can reuse the register $R = E_1$ and add the three lanes of the sheet at $x = 1$ so that E_1 cancels out and R correctly gets the parity before θ. One proceeds similarly to compute E_3 then E_0, each time re-using the register that contained the previous lane of the θ-effect.

4.2 Benchmarks

We implemented Xoodoo and Xoofff, and we benchmarked them on different processors: the 32-bit processors ARM Cortex-M0 and M3 and two mainstream desktop Intel processors, with the Skylake and SkylakeX architectures. The difference between these last two processors is that Skylake supports 256-bit vector instructions (AVX2), while SkylakeX offers 512-bit vector instructions (AVX-512).

On the Cortex-M3, we fit the state in 12 registers and use 2 registers for temporary variables. Furthermore, we can use the free rotations that this platform provides so that no explicit rotation instruction needs to be used. The state gets globally rotated, and this can be corrected at the end, as done by Schwabe et al. [SYY12]. With this technique, one round takes 49 cycles, so about 1.02 cycles/byte per round, plus 12 cycles for the global correction at the end. In Table 4, we assume that this global correction and the function call overhead are amortized on 18 rounds, as this can be done in the expansion phase of Xoofff.

The Cortex-M0 limits bitwise logical operations to the first 8 registers and does not support free rotations. This translates into more transfers between the first and the last registers, as well as explicit rotations. This trend is visible in Table 4, where we display the computational effort for one round of Xoodoo, as well as the round of other permutations.

Xoodoo's internal parallelism can be exploited with vector instructions, where one plane fits in a 128-bit register and several operations can be done plane-wise. Naturally, vector instructions are particularly well-suited to the parallelism provided by Xoofff's Farfalle construction. Here the size of the vector instruction determines the maximum number of Xoodoo instances that can be computed in parallel: 8 for Skylake (AVX2) and 16 for SkylakeX (AVX-512). In addition, the AVX-512 instruction set allows arbitrary three-input bitwise logical operations in one instruction, as well as rotations. The former is used to implement χ and the parity computation in θ, and the latter speeds up θ, ρ_{east} and ρ_{west}. This explains why Xoodoo and Xoofff are faster on SkylakeX than on Skylake even at the same level of parallelism. We present benchmarks of Xoofff in Table 5 with simple use-cases.

Table 4: Performance of a round of different permutations on Cortex-M0 and M3

	width	cycles/byte per round	
		ARM	
	bytes	Cortex M3	Cortex M0
Keccak-p[1600]	200	2.44	3.64
ChaCha [Ber08a]	64	0.69	2.00
Gimli	48	0.91	2.04
Xoodoo	48	1.10	3.76

Table 5: Performance of Xoofff on different platforms. The AES figures on Cortex-M0 and -M3 come from [Log, SS]. Skylake refers to an Intel® CoreâĎć i5-6500 processor at 3.2GHz and SkylakeX to an Intel® CoreâĎć i7-7800X processor at 3.5GHz. In both cases, the benchmark was performed with the Turbo Boost feature disabled.

	ARM Cortex		**Intel**		
use case	**M0**	**M3**	**Skylake**	**SkylakeX**	
Xoofff, 47-byte input and 48-byte output					
mask derivation only	1985	781	168	74	cycles
full deck function	5658	2568	504	358	cycles
Xoofff MAC					
512-byte input	18232	6846	1301	712	cycles
long input	26.0	8.8	0.90	0.40	c/byte
Xoofff Stream cipher					
512-byte output	17799	6510	1185	770	cycles
long output	25.1	8.1	0.94	0.51	c/byte
AES-128 counter mode	121.4	33.2	0.65	0.65	c/byte

5 Design rationale of Xoodoo

Xoodoo is an iterated permutation that is strongly inspired by Keccak-p: it is bit-oriented and its round function uses similar operations, see Section 5.1. It prevents high-probability differential trails and high-correlation linear trails by adopting the wide trail strategy, see Section 5.2. We discuss in Section 5.3 that Xoodoo enjoys the benefits of *weak alignment* like Keccak-p does. These benefits include negligible trail clustering of trails in differentials or correlations and the inapplicability of classes of attack such as truncated differentials [BDPA11a]. Moreover, the Xoodoo avalanche characteristics we report in Section 5.4 show that the combination of wide trail with weak alignment results in very fast diffusion. We discuss the high degree of symmetry of the round function in Section 5.5 and the choice of round constants that remove that symmetry in Section 5.6. Finally, in Section 5.7 we explain how we gravitated to the Xoodoo round function structure starting from that of Keccak-p and Section 5.8 reports on how we arrived at the choices of the rotation constants.

5.1 The Xoodoo round function in a nutshell

The Xoodoo round function uses the five step mappings θ, ρ_{west}, ι, χ and ρ_{east}. Four of them are very symmetric, as they operate on bits of the state in *the same way*, independently of their position. More formally, we say that a given step α is *translation-invariant* over a direction (x, y, z) if it commutes with a translation by (x, y, z), i.e., if $\alpha \circ \tau_{(x,y,z)} = \tau_{(x,y,z)} \circ \alpha$, where $\tau_{(x,y,z)}$ is a (cyclic) translation of the state by (x, y, z).

The nonlinear layer χ is an instance of the transformation χ that was already described and analyzed in [Dae95]. In Xoodoo it operates in parallel on 3-bit columns and as such forms a layer of 4×32 3-bit S-boxes. In general, χ has algebraic degree two [Dae95, Section 6.9], with interesting consequences for the analysis (see Section 5.2.3). For 3-bit units, χ is involutive and hence this also holds for its inverse. Consequently, r rounds of Xoodoo or its inverse cannot have an algebraic degree higher than 2^r. The χ layer is translation-invariant in all directions.

The mixing layer θ is a column parity mixer [SD18] that operates as follows. It builds the *parity plane* by adding the three planes and computes from this the θ*-effect* plane by cloning it, shifting these two copies and adding them. Then it adds the θ-effect to each of the three planes. The θ layer is invertible, has order 32 and its inverse is dense. Similar to χ it is translation-invariant in all directions.

The dispersion layer in Xoodoo consists of two steps. As in both the parity plane computation in θ and in χ the state bits interact only within columns, there is a need for dislocating the bits of the columns between every application of θ and of χ. For that reason, after each χ layer, we have the so-called *Eastern shift* ρ_{east} and after each θ layer the *Western shift* ρ_{west}. Both shift the planes, treating them as rigid objects. Clearly, ρ_{east} and ρ_{west} are translation-invariant in all directions with $y = 0$, so all *horizontal* directions. Their breaking up of the columns results in weak alignment [BDPA11a].

The translation-invariance in horizontal directions of the step mappings results in high symmetry. We destroy this symmetry by the classical addition of round constants in the step ι. In trail search the round constants can be ignored and we can take advantage of this high amount of symmetry.

The round function is defined as $\rho_{\text{east}} \circ \chi \circ \iota \circ \rho_{\text{west}} \circ \theta$. For the study of propagation, we often rephase the rounds as starting with ρ_{east} and ending in χ and group the sequence of linear mappings in $\lambda = \rho_{\text{west}} \circ \theta \circ \rho_{\text{east}}$, so the re-phased round function becomes $\chi \circ \iota \circ \lambda$. We call λ the *linear layer*. When studying propagation through the rounds, we denote the input to λ of the first round by a_0, its output b_0, the output of χ by a_1 and so on. One can think of b as before χ and a as after χ. The values of a and b with the same index i are connected through λ.

The rounds numbering starts from negative numbers with the last round having index 0. The reason for this is to avoid slide-like attacks when Xoodoo instances are used with different numbers of rounds in a single construction.

5.2 Difference and correlation propagation

In this section, we report on the difference and correlation propagation in Xoodoo. For a detailed description of the trail search we refer to Section 7.

5.2.1 Differential probability and trails

In many use cases we are interested in the differential propagation probabilities (DP) of a cryptographic primitive. In the case of Xoodoo specifically this is essential for the security of the compression phase of Xoofff. In particular, we would like to characterize the distribution of $\mathrm{DP}(\Delta_{\mathrm{in}}, \Delta_{\mathrm{out}})$ values over all input differences Δ_{in} and output differences Δ_{out} of our permutation, where $\mathrm{DP}(\Delta_{\mathrm{in}}, \Delta_{\mathrm{out}})$ is the fraction of input pairs with difference Δ_{in} that results in a difference Δ_{out} after the primitive. For iterated cryptographic permutations and block ciphers, this is a hard problem. However, we can gain understanding by studying *differential trails*, as the DP of a differential is the sum over the DP values of its trails.

An n-round differential trail is the concatenation of n round differentials (a_i, a_{i+1}) and is fully specified by the sequence $(a_0, a_1, \ldots, a_n)$. We say a pair (α, β) follows a trail when its initial difference is a_0 and the difference after round i is a_i for all $i \leq n$. We apply the rephasing introduced in Section 5.1 and use a redundant representation of trails, where we also include the differences after the linear layer: $(a_0, b_0, a_1, b_1 \ldots, b_{n-1}, a_n)$.

Clearly $b_i = \lambda(a_i)$ and each differential (b_{i-1}, a_i) over χ imposes a number of conditions on the members of the pair (α, β). We call this number the restriction weight $\mathrm{w_r}$. It follows that the restriction weight of a trail is the sum of the restriction weights of its round differentials. The restriction weight allows approximating the DP of a trail: If the conditions are independent, the DP of the trail is $2^{-\mathrm{w_r}}$. We report on the distribution of trail weights for 3 rounds in Table 6 at the end of this section.

Even in the absence of low-weight trails, one may have differentials $(\Delta_{\mathrm{in}}, \Delta_{\mathrm{out}})$ with high DP if there are very many differential trails from Δ_{in} to Δ_{out}, or if there are differential trails where the DP is much higher than $2^{-\mathrm{w_r}}$ due to dependencies between the round differential conditions. The study of these two aspects is closely related to that of *alignment* that we treat in Section 5.3.

A differential over χ is only possible if b_i and a_{i+1} have the same column activity pattern, i.e., the set of active columns must be the same. As shown in Section 5.2.3 below, the restriction weight equals twice the number of active columns in b_i, or equivalently, in a_{i+1}. It follows that the restriction weight of an n-round trail $(a_0, b_0, a_1, b_1, \ldots, b_{n-1}, a_n)$ is fully determined by the sequence $b_1, \ldots, b_{n-1}$ and is given by $\mathrm{w_r}(a_1) + \sum_{1 \leq i < n} \mathrm{w_r}(b_i)$. We call such a sequence a *differential trail core*, as in [DA12]:

$$Q = a_1 \xrightarrow{\lambda} b_1 \xrightarrow{\chi} a_2 \xrightarrow{\lambda} b_2 \xrightarrow{\chi} a_3 \xrightarrow{\lambda} \ldots a_{n-1} \xrightarrow{\lambda} b_{n-1} \; .$$

A trail core can be extended to an n-round differential trail by pre-pending a couple a_0, b_0 with b_0 compatible through χ with a_1 and appending a value a_n compatible through χ with b_{n-1}. It follows that a trail core Q represents in total $2^{\mathrm{w_r}(a_1)} \times 2^{\mathrm{w_r}(b_{n-1})}$ trails, all with the same weight. In our analysis, we bound the weight of trail cores. In the sequel, we use $\mathrm{w}(\cdot)$ as a shortcut notation for $\mathrm{w_r}(\cdot)$ when clear from the context.

5.2.2 Correlation and linear trail cores

Similarly to differential probability, we are interested in the input-output correlation properties of a cryptographic primitive f. In particular, we would like to characterize the distribution of $C(u_{\text{out}}^{\top} f(x), u_{\text{in}}^{\top} x)$, i.e., the correlation between linear combinations of output bits $u_{\text{out}}^{\top} f(x)$ and linear combination of input bits $u_{\text{in}}^{\top} x$ over all values of output (linear) mask u_{out} and input (linear) mask u_{in}. For iterated cryptographic permutations and block ciphers, this is a hard problem. Here too, we can gain understanding by studying *linear trails*, as a correlation $C(u_{\text{out}}^{\top} f(x), u_{\text{in}}^{\top} x)$ is the sum over the (signed) correlation contributions of its trails. In the sequel we will for readability slightly abuse terminology by speaking about correlations between masks.

An n-round linear trail is the concatenation of n single-round correlations. A correlation over round i is defined by a mask a_i at its output and a mask a_{i+1} at its input and we denote its correlation value $C(a_i, a_{i+1})$.

As for differential trails, we use a redundant representation by including the masks after the linear layer. To make notation consistent with differential trails, we rephase the rounds as starting with χ and ending with λ. However, as linear propagation is studied naturally from the output to the input, the trail first encounters λ and then χ of each round. A mask a_i at the output of λ maps to a mask $b_i = \lambda^{\top}(a_i)$ before λ. In this way a_i fully determines b_i via the linear layer λ. Our trails look like $(a_0, b_0, a_1, b_1 \ldots, b_{n-1}, a_n)$, where a_0 is the mask after the last round and a_n the mask before the first round. Note that the transposition denotes the following operation of a linear mapping: When the linear mapping μ is expressed as the multiplication by the matrix M, the transpose of μ, or $\mu^{\top}$, is a linear mapping given by the multiplication by $M^{\top}$. It follows that $\lambda^{\top} = \rho_{\text{east}}^{\top} \circ \theta^{\top} \circ \rho_{\text{west}}^{\top} = \rho_{\text{east}}^{-1} \circ \theta^{\top} \circ \rho_{\text{west}}^{-1}$ since the inverse of a bit transposition matrix is its transpose.

The correlation contribution of a linear trail is the product of its round correlations. Similarly to differential trails, we define a correlation weight for a round correlation $\mathrm{w_c}(b_i, a_{i+1})$, as $C^2(b_i, a_{i+1}) = 2^{-\mathrm{w_c}(b_i, a_{i+1})}$, and we define the weight of a trail as the sum of the weights of its round correlations. Also here we may have large input-output correlations $(u_{\text{out}}, u_{\text{in}})$ even in the absence of low-weight trails if there are very many linear trails from u_{out} to u_{in} and their signed correlation contributions combine constructively. This is again covered by our treatment of *alignment* in Section 5.3.

The correlation weight of a round correlation (a_i, a_{i+1}) is determined by the correlation weight of the mask couple (b_i, a_{i+1}) over χ. This correlation is only non-zero if b_i and a_{i+1} have the same column activity pattern and, as shown in Section 5.2.3 below, the correlation weight equals twice the number of active columns in b_i, or equivalently, in a_{i+1}. It follows that the correlation weight of an n-round trail $(a_0, b_0, a_1, b_1, \ldots, b_{n-1}, a_n)$ is fully determined by the sequence $b_1, \ldots, b_{n-1}$ and is given by $\mathrm{w_r} a_1 + \sum_{1 \leq i < n} \mathrm{w_r} b_i$. This is a *linear trail core*.

$$Q = a_1 \xrightarrow{\lambda^{\top}} b_1 \xrightarrow{\chi} a_2 \xrightarrow{\lambda^{\top}} b_2 \xrightarrow{\chi} a_3 \xrightarrow{\lambda^{\top}} \ldots a_{n-1} \xrightarrow{\lambda^{\top}} b_{n-1} \ .$$

We also use $\mathrm{w}(\cdot)$ as shortcut notation for $\mathrm{w_c}(\cdot)$ when clear from the context that we are dealing with a linear trail.

5.2.3 Properties of χ

χ can be defined generically, operating on n bits arranged in a circle. The differential and correlation propagation properties of this generic χ are non-trivial and have been described in [Dae95, Section 6.9]. Thanks to the fact that in Xoodoo χ operates on 3-bit circle, formed by the columns, propagation of differences and masks through it can be specified very compactly. We do this in Proposition 1.

Proposition 1. *At column-level, a non-zero difference $b = (b_0, b_1, b_2)$ at the input and a non-zero difference $a = (a_0, a_1, a_2)$ at the output of χ are compatible if $a \cdot b$ has odd parity, or equivalently $a_0b_0 + a_1b_1 + a_2b_2 = 1$. Likewise, a non-zero mask $b = (b_0, b_1, b_2)$ at the output and a non-zero mask $a = (a_0, a_1, a_2)$ at the input of χ are compatible if $a \cdot b$ has odd parity.*

We do not give a proof for this proposition as it can easily be checked exhaustively. Proposition 1 has several corollaries:

Corollary 1. *For fixed (difference or mask) a, the compatible (difference or mask) b values form an affine space of dimension 2 and vice versa.*

Corollary 2. *The restriction weight of a differential over χ is equal to two times the number of active columns in b, or equivalently in a.*

Corollary 3. *The correlation weight of a differential over χ is equal to two times the number of active columns in b, or equivalently in a.*

These three corollaries simplify trail analysis in comparison with that of Keccak-p. Thanks to the first one both forward and backward extension can make use of linear algebra. Thanks to the last two corollaries we can replace the weight and the minimum reverse weight (see [DA12]) by the number of active columns.

5.2.4 Trail weight distributions

As we will detail in Section 7, we have determined all 3-round trails up to weight 50, both for linear and differential trails and we list them in Table 6. The minimum weight for both types of trail is 36. The trails of weight 36 and 38 are simply due to the 3-plane structure of Xoodoo and are described in Section 7.6. Apart from them, there are no 3-round trails below weight 44.

Table 6: The 3-round trail cores. The number of trail cores is up to translations along x and z, see Section 5.5.

Weight	# differential	# linear
36	4	4
38	3	3
44	3	5
46	24	24
48	31	29
50	55	56

When extending the 3-round trails to 6 rounds, none were found. This results in a lower bound on the weight of 6-round trails is $2T_3 + 4 = 104$, for both differential and linear trails. The lower bounds on trail weights are summarized in Table 7.

Table 7: The weight of the best differential and linear trails (or lower bounds) as a function of the number of rounds.

# rounds:	1	2	3	4	5	6
differential:	2	8	36	≥ 54	≥ 56	≥ 104
linear:	2	8	36	≥ 54	≥ 56	≥ 104

5.3 Alignment

In [BDPA11a], Bertoni et al. investigated an aspect of round functions called *alignment*. Alignment is related to the propagation of *activity patterns* through the linear layer of a round function. We summarize what alignment means in the context of round functions that have an S-box layer as non-linear layer. The s-bit S-boxes partition the bits of the state in subsets that are processed by the same S-box, and we call those *boxes*. When applying a difference Δ, a mask u or in general any state-sized binary pattern, we can define its corresponding *(box) activity pattern*. The activity pattern corresponding to a concrete pattern specifies for each box whether it contains only zero bits or at least one bit with value 1. In the former case we call the box *passive*, in the latter we call it *active*. Given an activity pattern $\overline{a}$ with n active boxes, there are $(2^s - 1)^n$ concrete patterns a compliant with $\overline{a}$. We will treat activity patterns as sets of concrete patterns and we say $a \in \overline{a}$. Clearly, an invertible S-box layer preserves (box) activity patterns of differences and of linear masks (and those of most other propagating structures). This is not true in general for a linear layer.

If the linear layer maps the elements of $\overline{a}$ to many different activity patterns, we say it has *weak alignment*. Otherwise, if it maps large fractions of the elements of $\overline{a}$ to a small set of activity patterns, we speak of *strong alignment*. This can be applied to different types of patterns but the most important are differences and (linear) masks. Their propagation through the linear layer is governed by different laws but they behave in very similar ways (see Sections 5.2.1 and 5.2.2). We denote by $\lambda(\overline{a})$ the set of states b with $b = \lambda(a)$ and $a \in \overline{a}$.

Let us illustrate strong and weak alignment by applying the simplest difference activity patterns to two of the best known cryptographic primitives: Rijndael [DR02] and Keccak-p [BDPA11b]. The simplest activity pattern is that with a single active box:

- In Rijndael, boxes are bytes and the linear layer is `MixColumns` ∘ `ShiftRows`. ShiftRows is a transposition of boxes and therefore moves inputs with equal activity patterns to outputs with equal activity patterns. This allows us to ignore it and focus on `MixColumns`. If we apply an input to the `MixColumns` matrix with a single active byte, all 4 output bytes will be active. This is a consequence of the fact that the `MixColumns` matrix has branch number 5. So the Rijndael linear layer maps all $2^8 - 1 = 255$ patterns with a single active byte at some position to the same 4-byte activity pattern, and this is the case for all 16 byte positions of the active byte at the input.

- In Keccak-$p[400]$, boxes are 5-bit rows and the linear layer $\lambda = \pi \circ \rho \circ \theta$ maps the 31 single-row patterns to 31 different activity patterns, and this for all row positions [BDPA11a].

Clearly, for this simple case Rijndael has the strongest possible alignment and Keccak-$p[400]$ the weakest possible. If we consider input activity patterns with multiple active boxes the distinction is less extreme but the trend is similar.

Strength of alignment manifests itself in the number of trail cores in *truncated differentials* [Knu94]. A truncated differential is defined by a couple of activity patterns $(\overline{\Delta_{\text{in}}}, \overline{\Delta_{\text{out}}})$ and is the set of all differentials with input difference in $\overline{\Delta_{\text{in}}}$ and output difference in $\overline{\Delta_{\text{out}}}$. A trail is in $(\overline{\Delta_{\text{in}}}, \overline{\Delta_{\text{out}}})$ if its initial difference is in $\overline{\Delta_{\text{in}}}$ and its final difference is in $\overline{\Delta_{\text{out}}}$. Let us now consider a truncated differential $(\overline{\Delta_{\text{in}}}, \overline{\Delta_{\text{out}}})$ over $\chi \circ \lambda \circ \chi$. The number of trail cores in this truncated differential is the number of elements in $\lambda(\overline{\Delta_{\text{in}}}) \cap \overline{\Delta_{\text{out}}}$. In the case of weak alignment the elements of $\lambda(\overline{\Delta_{\text{in}}})$ will per definition have many different activity patterns and hence for any $\overline{\Delta_{\text{out}}}$, the set $\lambda(\overline{\Delta_{\text{in}}}) \cap \overline{\Delta_{\text{out}}}$ will be small. This implies that truncated differentials, and a fortiori ordinary differentials, will have a small number of trail cores. In the case of strong alignment $\lambda(\overline{\Delta_{\text{in}}}) \cap \overline{\Delta_{\text{out}}}$ may be large and (truncated)

differentials possibly have many trails. An analogous reasoning holds for correlations and linear trails, where clustering depends on the alignment of $\lambda^\top$ instead.

With a similar (but not the same) reasoning it can be shown that in the case of weak alignment the conditions imposed by round differentials in a trail tend to be independent, while strong alignment increases the risk for dependence, as observed in plateau trails of Rijndael [DR07].

In Xoodoo the boxes are 3-bit columns and an activity pattern has the shape of a plane. Clearly, the linear layer maps the $2^3 - 1$ single-row patterns (both differences and masks) to 7 different output activity patterns. This is weak alignment. We experimented with randomly generating many activity patterns $\overline{a}$ with $n \leq 10$ active columns and for the vast majority of cases the 7^n elements of $\lambda(\overline{a})$ had 7^n different activity patterns.

5.4 Avalanche behavior

When reporting on (reduced-round) cryptographic functions, one often mentions criteria such as *full diffusion*, *avalanche* and *strict avalanche* criterion (SAC) [WT85]. These criteria are useful in estimating the vulnerability of the cipher to certain attacks. Each of these criteria is binary: it is either met, or it is not. Typically, for an iterated cipher one reports on the number of rounds required to satisfy it. In this section we define metrics that allow evaluating in a more fine-grained way how the function realizes it through the rounds.

5.4.1 Definition of avalanche metrics

Concretely, we compute the *avalanche probability vector* of a cryptographic primitive F for some input difference Δ: a vector $P_{\Delta F}$ where component i is the probability that bit i of the output of F flips due to the input difference Δ. For clarity, we specify the generation of the avalanche probability vector in Algorithm 2. After M samples, the expected standard deviation of the elements of $P_{\Delta F}$ is $1/\sqrt{M}$. So for high precision, M must be chosen large enough. In our experiments, we took $M = 250000$.

Algorithm 2 Computation of the avalanche probability vector $P_{\Delta F}$.

Parameters: a transformation F over $\mathbb{Z}_2^b$, an input difference Δ and number of samples M.
Output: the avalanche probability vector $P_{\Delta F}$.
Initialize a b-bit vector p of probabilities p_i to all zeroes
for M randomly generated states A **do**
  Compute $B = F(A) + F(A + \Delta)$
  for all state bit positions i **do**
    $p_i = p_i + B_i/M$
return $P_{\Delta F} = p$

From the avalanche probability vector $P_{\Delta F}$ we extract three metrics, each one measuring an aspect of the difference at the output of F due to a given input difference Δ. In the following we write p_i for $P_{\Delta F}[i]$.

Avalanche dependence: number of output bits that may flip, defined as:

$$D_{\text{av}}(F, \Delta) = b - \sum_i \delta(p_i) \ ,$$

with $\delta(x)$ equal to 1 if $x = 0$ and 0 otherwise. This metric generalizes full diffusion, that is satisfied if $D_{\text{av}}(F, \Delta) = b$ for all Δ with Hamming weight 1.

Table 8: Avalanche scores

stage	δ_a D_{av}	δ_a $\overline{w}_{\text{av}}$	δ_a H_{av}	δ_K D_{av}	δ_K $\overline{w}_{\text{av}}$	δ_K H_{av}	δ_b D_{av}	δ_b $\overline{w}_{\text{av}}$	δ_b H_{av}
a_{-2}	384	191.999	383.999	384	191.963	383.999	384	191.976	383.999
b_{-2}	381	187.636	357.483	384	189.751	376.877	384	191.977	383.999
a_{-1}	293	176.502	224.000	346	183.942	315.999	384	191.991	383.999
b_{-1}	3	2.000	2.000	6	3.999	4.000	279	168.504	220.999
$\mathbf{a_0}$	**1**	**1.000**	**0.000**	**2**	**2.000**	**0.000**	133	133.000	0.000
$\mathbf{b_0}$	7	7.000	0.000	**2**	**2.000**	**0.000**	**1**	**1.000**	**0.000**
a_1	21	14.001	14.000	6	3.999	4.000	3	1.998	2.000
b_1	102	64.494	75.000	42	28.005	28.000	21	13.991	14.000
a_2	210	94.748	187.205	105	48.497	87.788	63	28.003	50.722
b_2	371	181.096	366.199	293	140.463	268.162	207	94.992	182.936
a_3	384	188.576	382.609	357	164.850	343.225	321	128.604	293.368
b_3	384	191.997	383.999	384	191.938	383.936	384	188.014	381.672

Avalanche weight: expected Hamming weight of the output difference, defined as:

$$\overline{w}_{\text{av}}(F, \Delta) = \sum_i p_i \ .$$

Clearly $\overline{w}_{\text{av}}(F, \Delta) \leq D_{\text{av}}(F, \Delta)$. This metric generalizes the avalanche criterion, that is satisfied if $\overline{w}_{\text{av}}(F, \Delta) \approx b/2$ for all Δ with Hamming weight 1.

Avalanche entropy: uncertainty about whether output bits flip, defined as an entropy:

$$H_{\text{av}}(F, \Delta) = \sum_i \left(-p_i \log_2(p_i) - (1 - p_i) \log_2(1 - p_i)\right) \ .$$

This metric generalizes SAC, that is satisfied if $H_{\text{av}}(F, \Delta) \approx b$ for all input differences Δ with Hamming weight 1.

The three metrics have values in the range $[0 \dots b]$ and for a random transformation F we have for any input difference Δ: $D_{\text{av}}(F, \Delta) \approx b$, $\overline{w}_{\text{av}}(F, \Delta) \approx b/2$ and $H_{\text{av}}(F, \Delta) \approx b$.

5.4.2 Reporting on the avalanche properties of Xoodoo

We report on the performance of Xoodoo with respect to the three avalanche metrics described above, for particular input differences and for different number of rounds and inverse rounds. For each of these metrics, we report on the worst-case values of these metrics: the minimum value taken over all individual input differences of given type. We give the results in Table 8.

In Table 8, we follow the convention a_i and b_i introduced in Section 5.2.1 and we apply the difference in the stage a_0. As $b_i = \lambda(a_i)$ with λ linear, applying a difference Δ at a_0 is equivalent to applying a difference $\lambda(\Delta)$ at b_0. Clearly, the avalanche scores at stage a_i report on the difference after i Xoodoo rounds. If i is positive, these are forward rounds, if i is negative, these are inverse rounds. Avalanche scores at stages b_i add one linear layer λ to it. For the differences applied at a_0 or b_0, we consider:

δ_a Single-bit differences at a_0.

δ_K Orbitals at the input/output of θ. These are 2-bit differences both at a_0 and b_0.

δ_b Single-bit differences at b_0.

The avalanche behavior of a cipher gives a good indication of the number of rounds that certain structural distinguishers can cover. For example, as a rule of thumb, it is hard to find impossible differentials that span a number of rounds that is more than two times the number of rounds it takes to have full diffusion. From Table 8 we can see that strict avalanche is reached after 3.5 rounds in forward direction and after 2 rounds in backward direction.

5.5 Symmetry

A plane in Xoodoo can be seen as an infinite state periodic in two directions: period 4 in the direction of the x-axis and period 32 in the direction of the z axis. Put otherwise: it is invariant for translations over any vector in the two-dimensional lattice with basis vectors $(4, 0)$ and $(0, 32)$. We express this lattice as $\langle(4, 0), (0, 32)\rangle$ and we call this the Xoodoo lattice Ξ. Differences and masks propagate irrespective of the round constants so that symmetry can be maintained during propagation.

This effect also exists in Keccak-p and is called Matryoshka [BDPA11b]: states (differences or masks) of Keccak-$p[25 \times 2^n]$ with symmetry $\forall(x, y, z) : A[x, y, z] = A[x, y, z+2^j]$ map to states of Keccak-$p[25 \times 2^{n-j}]$. The invariance of Xoodoo with respect to all horizontal translations results in a two-dimensional Matryoshka property. A symmetric state of Xoodoo can be expressed with respect to a lattice V: $\forall(x, y, z)$ and $v \in V$: $A[x, y, z] = A[(x, y, z) + v]$. If we take V the Xoodoo lattice Ξ, this describes a regular Xoodoo state. If V is a lattice that has Ξ as a sub-lattice, we have a state with additional symmetry. Each symmetric state maps to a state in a smaller instance of Xoodoo, with equal steps χ and variants of ρ_{west}, ρ_{east} and θ.

Each lattice V that has Ξ as a sub-lattice defines a symmetry class S_V that forms a subset of the state values. A state a is in the symmetry class S_V if it is invariant with respect to any translation along V and there exists no lattice V' with $V \subset V'$ such that a is invariant with respect to V'. The symmetry classes form a partition of the state space.

We can exhaustively specify the symmetry classes by the basis of their lattices, where the first element of the basis is of the form $(0, 2^e)$ with $0 \leq e \leq 5$. For a basis with first vector $(0, 2^e)$, the second vector is in the following range:

- $(4, 0)$, $(2, 0)$ and $(1, 0)$: exists for all e
- $(2, 2^{e-1})$ and $(1, 2^{e-1})$: exist for $e > 0$
- $(1, 2^{e-2})$ and $(1, 2^{e-2}3)$: exist for $e > 1$

We count here $6 \times 3 + 5 \times 2 + 4 \times 2 = 36$ lattices, including Ξ itself. The symmetry class S_V of the lattice $V = \langle(1, 0), (0, 1)\rangle$ can further be sub-divided in 2 symmetry classes due the fact that all-0 or all-1 planes give rise to shift-invariance along the y axis. The two classes are the one with three equal planes or the one with different planes. We can model this split by extending the lattice vectors by a y-component and adding a third lattice vector. The 3-equal-plane lattice can now be specified as $\langle(1, 0, 0), (0, 0, 1), (0, 1, 0)\rangle$. The 36 other lattices just get the additional vector $(0, 3, 0)$. For readability, we will stick to the two-dimension representation and will ignore the y component. So there are 37 symmetry classes in total. For 36 of these classes, the elements exhibit some symmetry within the boundaries of the Xoodoo state. For one class this is not the case: namely S_Ξ. This class contains about $2^{384} - 3 \times 2^{192}$ of the 2^{384} state values, so the vast majority.

5.6 Round constants

Thanks to their shift-invariance and invertibility, applying any step mapping of the round function other than ι to a state in a symmetry class S_V results in a state in the same

symmetry class S_V. The symmetry classes are hence invariant subsets [LAAZ11]. Moreover, the union of any subset of the 37 symmetry classes is also an invariant subset. As n disjoint subsets can be grouped into two non-empty subsets in $2^{n-1} - 1$ ways, all steps except ι have the same $2^{36} - 1$ invariant subsets. This property would carry over to a round function variant without ι and thus to such a Xoodoo$[n_\mathrm{r}]$ variant irrespective of the number of rounds.

We chose the round constants to destroy shift-invariance of the round function and to remove all these $2^{36} - 1$ invariant subsets. For this reason, we chose them to be in S_Ξ so that addition of a round constant maps any state not in S_Ξ to a state in S_Ξ. As such the round function cannot have any of the $2^{36} - 1$ subsets as invariant subsets. As S_Ξ contains more than half of the state values, it is not possible to group the symmetry classes into two equal subsets. This allows us to exclude the case of a subset mapping to its complement.

It may be the case that the effect of the round constant in two (or more) consecutive rounds would compensate each other. For that reason, we have opted for round constants with support in a single lane $(x, y) = (0, 0)$ so that the subsequent application of θ will make it propagate to other lanes. Moreover, to avoid attacks that exploit equality of the rounds such as slide attacks [BW99], the round constants depend on the round number.

Naturally, Xoodoo[6] is a permutation and for any permutation the union of the elements in any subset of its cycles forms an invariant subset. For a random permutation these invariant subsets would not carry enough structure to be exploitable in Farfalle. Investigating what are the relevant cycle behavior properties of a permutation in the context of Farfalle and whether such behavior is present in Xoodoo[6] are interesting topics for future research.

Finally, for allowing efficient implementation on ARM Cortex-M3 processors, the round constants span at most 4 consecutive positions along the z axis.

5.7 The making of Xoodoo

For the design of Xoodoo, we started from Keccak-p and aimed for a 384-bit permutation. We decided to use a nonlinear layer similar to Keccak-p's χ but on 3 bits instead of 5 to match the factor 3 in 384. Anyway, χ needs to apply on an odd number of bits, otherwise it is not invertible. Another option would be to aim for a 320-bit permutation and to stick to Keccak-p's χ on 5 bits, but we thought that 384 bits would be better suited.

Next, we opted for 32-bit lanes. Another option could have been to take 128-bit lanes and rewrite the algorithm into equivalent operations on 32-bit words using the bit interleaving technique [BDP+12], but we found that a structure with 3 planes of 4×32 bits was easier to describe.

For the mixing layer, we opted for a column parity mixer, similar to Keccak-p's θ. For this layer to have a dense inverse [SD18], θ needs to work on columns of odd size. This made it clear that both χ and θ would need to work on 3-bit columns.

For dispersion, Keccak-p uses two operations: π that moves lanes and ρ that translates lanes, both applied before χ, but it has no dispersion layer between χ and θ. In the case of Xoodoo, however, we need two dispersion layers, to avoid overlaps between the mixing and the nonlinear layers: one before χ and one before θ. So the idea of using ρ_east and ρ_west came early in the design. Another option would have been to have χ operate on skewed columns, but this was equivalent and just more complicated to describe than the two dispersion layers. Moreover, it seemed that a ρ mapping that shifted planes rather than lanes would be sufficient, and so would be simpler than that in Keccak-p. We definitely liked the idea of three independent rigid objects interacting through χ and θ.

Initially, we thought that we could get away with one of the dispersion layers that moves only along the x axis, i.e., to only move lanes without any shifts, hence saving on the number of rotations in the implementation. However, it turned out that this was not

enough, and we therefore added some translation along z to make the work more balanced between the two ρ steps.

Finally, we fixed all the rotation offsets as described in the next section.

5.8 Choosing the shift offsets in the light of trails

Once we defined the general structure of the Xoodoo, we set out experiments to find good shift offsets for the linear layer. Specifically, the family we investigated is as in Algorithm 1, where

- the θ-effect is computed as $E \leftarrow P \lll (1, t_1) + P \lll (t_3, t_2)$,
- in ρ_{west}, the translation of A_2 is $A_2 \leftarrow A_2 \lll (0, w_1)$, and
- in ρ_{east}, the translation of A_2 is $A_2 \leftarrow A_2 \lll (e_0, e_1)$,

for parameters t_1, t_2, t_3, w_1, e_0 and e_1.

In short, we chose the shift offsets of ρ_{east}, θ and ρ_{west} such that the number of trails of weight below 44 is minimized, both for differential and linear trails. Actually, only the so-called inherent trails remain below 44, see Section 7.6. Let us detail the decision process.

We restricted the offsets in ρ_{west} and ρ_{east} from the start to limit their computational cost. As only relative shifts count, we start by not shifting plane A_0 at all. For the other two planes, we set out to limit the number cyclic lane shifts as they are expensive on some platforms. In an early phase, we limited shifts in ρ_{west} and ρ_{east} to shifts along the x-axis (with an offset of the form $(s, 0)$) and shifts along the z-axis (of the form $(0, t)$). In particular, A_1 would undergo a shift $(1, 0)$ in ρ_{west} and a shift $(0, 1)$ in ρ_{east} and A_2 a shift $(0, w_1)$ in ρ_{west} and $(e_0, 0)$ in ρ_{east}. In that way, both $\rho_{\text{west}} \circ \rho_{\text{east}}$ and $\rho_{\text{east}} \circ \rho_{\text{west}}$ shift any pair of planes with respect to each other over an offset of the form (s, t) with $s \neq 0 \neq t$. However, our propagation experiments immediately revealed systematic low-weight trails. Adding a shift of A_2 along the z-axis in ρ_{west} made these trails go away, even with the offset $e_1 = 8$ that is cheaper on some platforms.

For θ, we needed to select the two offsets in the computation of the θ-effect. We decided to fix $t_3 = 1$, meaning that both affected columns are in the same lane. This allows computing the θ-effect with just one additional register (see Section 4.1).

Loops are sequences of 32 odd columns such that the θ-effect cancels out, see Section 7.3.2 for a more formal definition. Given the number of odd columns, the weight of a trail containing a loop is well above our target, so this was not considered a problem. Yet, we required that $t_1 - t_2$ is odd, as otherwise loops with fewer than 32 odd columns would exist.

There remained the choice of the values of t_1, t_2, w_1, e_0 in the light of differential and linear trails. For this part, we refer to Section 7 for the terminology. We proceeded in two steps. First, we looked for 3-round trail cores where the input to θ is in the kernel in both rounds. In this case, θ acts as the identity, and this process is thus independent of the values of t_1, t_2. This allowed us to select good candidates for w_1, e_0. Second, we extended the search to all 3-round trail cores to select good values for t_1, t_2.

1. The Vortex is an inherent trail core with weight 36 that is in the kernel (see Table 9). Other trails in the kernel we found were a few trails of weight $46 = 16 + 14 + 16$ and three trails of weight 48 (one with weight profile $18 + 16 + 14$, one symmetric along x with profile $24+16+8$, and one symmetric along z with profile $8+16+24$). We looked for couples (w_1, e_0) with $e_0 \in \{2, 3\}$ and $w_1 \in \{2, \ldots, 31\}$ such that no other trails in the kernel with weight up to 48 exist. We found generally better results for $e_0 = 2$, so we decided to fix that value, and the values $w_1 \in \{2, 3, 4, 5, 6, 7, 9, 10, 11, 15, 19, 20, 21\}$ satisfied the criterion.

2. Outside the kernel, i.e., without any constraints on the input of θ, the best trails are the Single-orbital fan and the θ^2-glide. We looked for tuples (t_1, t_2, w_1), with w_1 in the set above, such that no other trails below weight 44 would exist.

After this selection process, we were left with a list of about 20 candidate tuples for (t_1, t_2, w_1). We finally selected a single tuple from this list on the basis of performance with respect to avalanche criteria, as illustrated in Table 8.

6 Design rationale of Xoofff

In this section, we first discuss the rolling functions, then give a rationale for the number of rounds in the different permutations.

Both rolling functions operate as 12-stage feedback shift registers (FSR), with the lanes mapping to the 32-bit stages. We can define an infinite sequence V of stages V_i with $i \geq 0$. The initial state/mask consists of the first 12 stages and the state/mask after j iterations of the rolling function consists of stages j to $j+11$. The mapping of these 12 stages to the lanes of the state/mask at iteration t is as follows: $A_{x,y} = V_i$ with $i = t + y + 3x$. The first 12 stages V_0 to V_{11} are the initial value of the mask/state. All subsequent stages V_t are defined by a recursion of the type $V_t \leftarrow F(V_{t-1} \ldots V_{t-12})$. Clearly, this is the operation of an FSR.

6.1 The rolling function roll_{Xc}

The rolling function roll_{Xc} is a lightweight invertible linear FSR of maximum order operating on the entire 384-bit state constructed as proposed by Granger et al. [GJMN16]. It has the following minimal polynomial:

$$\begin{aligned}1 + x^{46} + x^{92} + x^{94} + x^{138} + x^{142} + x^{186} + x^{188} + x^{190} + x^{199} + x^{223}\\ + x^{238} + x^{245} + x^{247} + x^{269} + x^{271} + x^{284} + x^{286} + x^{295} + x^{319} + x^{330}\\ + x^{334} + x^{341} + x^{343} + x^{352} + x^{365} + x^{367} + x^{378} + x^{380} + x^{382} + x^{384} .\end{aligned}$$

As a consequence, each non-zero mask value will be in a cycle of length $2^{384} - 1$. The zero mask value is a fixed point in our rolling function. We think the probability that a user key K maps to such a mask value is negligible.

The recursion in the stage representation is:

$$V_{j+12} \leftarrow V_j + (V_j \ll 13) + (V_{j+1} \lll 3) .$$

As V_{j+12} depends only on V_j and V_{j+1}, this allows the parallel computation of up to 11 subsequent iterations. In other words, given that we have V_j to V_{j+11}, we can compute V_{j+12} to V_{j+22} in parallel.

An important purpose of roll_{Xc} is to avoid affine spaces of large dimensions. This aspect is discussed in more detail by Bertoni et al. in [BDH+17].

6.2 The rolling function roll_{Xe}

The rolling function roll_{Xe} is a lightweight invertible non-linear FSR. It is non-linear to resist against state-recovery attacks described in [CFG+18] that work if roll_{e} is linear, p_{b} is Keccak-p with 6 rounds and the adversary has a very long sequence of output blocks.

The recursion in the stage representation of roll_{Xe} is:

$$V_{j+12} \leftarrow (V_j \lll 5) + (V_{j+1} \cdot V_{j+2}) + (V_{j+1} \lll 13) + \texttt{0x00000007} . \quad (3)$$

As V_{j+12} depends only on V_j, V_{j+1} and V_{j+2}, this allows the parallel computation of up to 10 subsequent iterations, so given V_j to V_{j+11}, we can compute V_{j+12} to V_{j+21} in parallel.

The recursion contains a bitwise product of two stages for non-linearity, two linear terms for diffusion and a constant term to remove symmetry and avoid fixed points. The algebraic normal form (ANF) of $\mathrm{roll}^{i}_{\mathrm{Xe}}(A)$ is non-linear. Informally, the criterion for $\mathrm{roll}_{\mathrm{Xe}}$ is that the degree and number of monomials in this ANF grows sufficiently with i to thwart attacks like the aforementioned ones.

The ANF of $\mathrm{roll}^{i+1}_{\mathrm{Xe}}(A)$ can be computed iteratively with i. Every iteration, the non-linear term in (3) introduces 32 fresh products, contributing to the increase in algebraic degree. The linear terms and the constant contribute to the increase of the number of monomials. Due to the fact that bits of V_j are independent of V_{j-1} to V_{j-10}, monomial growth is relatively slow. Still, as the aforementioned attacks require a very long sequence of output blocks, we believe this is sufficient.

Since $\mathrm{roll}_{\mathrm{Xe}}$ is non-linear, its selection process is not as straightforward as that of $\mathrm{roll}_{\mathrm{Xc}}$. We applied the following tests to arrive at our choice:

Fixed points First we test for cycles of length one (i.e., fixed points), preferring candidates that have the fewest. Since a single iteration of the rolling function merely moves all of the lanes by one position and calculates a new lane, the necessary and sufficient condition for the state to be a fixed point is one where all lanes comprising the state, including the newly calculated lane, are equal. For 32-bit (and smaller) lane sizes, it is feasible to enumerate all fixed points simply by testing every possible lane value.

Short cycles Then we test for short cycles induced by various symmetric states, preferring candidates for which no cycles could be found. The tested states are:

- Alternating bit patterns 10101... and 01010...
- Single bit toggles such as 10000..., 01000..., etc. and complements 01111..., 10111..., etc.
- Single bit toggles like the above, but starting with alternating lanes as $0^{32}||1^{32}||0^{32}||$... and $1^{32}||0^{32}||1^{32}||$...

Note that having an invertible rolling function is desirable since it is conducive to longer cycles, and makes the short cycle test more efficient since it obviates the need to keep track of all states that have been seen. Instead, only the initial state needs to be remembered; if the state arrives back at the initial state, then a cycle has been found. In our tests, we ran 10^6 iterations per pattern and candidate.

Monomial count Then we run simulations to gain an understanding of the number of monomials present in the algebraic normal form of the rolling function after n iterations, preferring candidates that contain the greatest number of monomials in the fewest number of iterations. We repeat this simulation for monomials of degree two, three, four and greater. We describe the degree two monomial count test in Algorithm 3; the higher degrees are a generalization of this algorithm except with a random sampling of monomial coordinates to make the execution time more practical.

6.3 Addressing Farfalle and Kravatte attacks

In this subsection we discuss Xoofff in the light of the attack paths on Farfalle and Kravatte as identified in [BDH+17]. Specifically, our choice for the number of rounds in p_{b}, p_{c}, p_{d} and p_{e} follows essentially the same rationale as for the number of rounds in Kravatte Achouffe, with 6 rounds for the four permutations. Moreover, our choice of the two rolling functions was guided by the experience with Kravatte. We will follow the canvas of Sections 5 and 7.4 of [BDH+17].

Algorithm 3 Definition of monomials(F, r, M) for monomials of degree two.

Parameters: a b-bit non-linear rolling function F, number of rounds r and number of samples M
Let $\delta_i = 0^i||1||0^{b-i-1}$
Initialize monomial count p to 0
for $i = 0$ to $b-1$ **do**
  for $j = i+1$ to $b-1$ **do**
    for all state bit positions k **do**
      for M randomly generated states A **do**
        Compute $B = F^r(A) + F^r(A + \delta_i) + F^r(A + \delta_j) + F^r(A + \delta_i + \delta_j)$
        If $B_k = 1$, increment p and break out of innermost loop
return p

6.3.1 Accumulator collisions

Clearly, as the accumulator has width 384, collisions can be found generically with expected complexity 2^{192} Xoofff executions, the so-called birthday bound. Section 5.1 of [BDH+17] discusses three *non-generic* methods to achieve collisions.

The first two methods are finding sets of input blocks that contribute 0 to the accumulator and input block swapping that leave the accumulator unchanged. Both methods strongly depend on the properties of the mask rolling function. In [BDH+17, Section 5.1] an extensive analysis provides evidence that a maximum-order linear rolling function with a characteristic polynomial that is not too sparse has the properties to make these methods infeasible. This is the case of the mask rolling function roll_{Xc}. Note that in Kravatte the mask rolling function operates on 320 of the 1600-bit mask, thereby leaving 1280 bits untouched, while roll_{Xc} is a maximum-order linear mapping operating on the full 384-bit state.

The third method is the exploitation of differentials in p_{c} and the choice of Xoodoo[6] for p_{c} is actually motivated by this method. In this method one applies message pairs that differ in two blocks by the same difference Δ and a collision occurs iff the differences cancel in the accumulator. This happens with probability

$$\text{CP}(\Delta) = \sum_{\gamma} \text{DP}_f^2(\Delta, \gamma) \ .$$

We call $\text{CP}(\Delta)$ the collision probability of Δ. Applying n pairs with the same difference Δ gives success probability $n\text{CP}(\Delta)$. In general, if we apply a set of messages $\mathcal{X} = \{X^{(1)}, X^{(2)}, \dots X^{(n)}\}$, the success probability of having two message collide is $\text{CP}(\mathcal{X}) = \sum_{i,j} \text{CP}(X^{(i)} + X^{(j)})$. Finding a set $\mathcal{X}$ that maximizes its collision probability is in general a hard problem. Section 7.4.1 of [BDH+17] discusses how to find a set $\mathcal{X}$ and an estimate of $\text{CP}(\mathcal{X})$ for Kravatte based on the properties of Keccak-$p[6, n_{\text{r}}]$. When working out a similar analysis for Xoofff, we found a method to increase the success probability for generating collisions. We explain our method in the next subsection.

6.3.2 An improved collision generating method

We describe the method for a 6-round permutation, but it can trivially be generalized to any number of rounds. As in [BDH+17, Section 7.4.1], we make the assumption that the differentials over 6 and 5 rounds with the highest DP are dominated by a single trail. We believe this is the case for both of Xoodoo and Keccak-p thanks to weak alignment. Hence, in a pair of colliding messages, the differences Δ follow the same trail in both active blocks to end up in the same difference γ. Denoting trails solely by the differences b_i at the input of χ, (with $b_0 = \lambda(\Delta)$) the ones followed by the differences in the two active blocks

is $(b_0, b_1, b_2, b_3, b_4, b_5, \gamma)$. The value of γ is not important, as long as it is the same in both trails. There are $2^{\mathrm{w}(b_5)}$ possible values for γ, hence applying $M/4$ two-block message pairs with input differences Δ leads to following success probability:

$$M2^{-(2+2\mathrm{w}(b_0)+2\mathrm{w}(b_1)+2\mathrm{w}(b_2)+2\mathrm{w}(b_3)+2\mathrm{w}(b_4)+\mathrm{w}(b_5))} .$$

In [BDH+17, Section 7.4.1] this success probability is slightly increased by embedding the pairs in a larger structure exploiting multiple input differences that have a high CP. Our new method does something similar, but with a more spectacular increase of success probability: it removes the contribution of b_0 to the weight altogether.

In Xoofff, we arrange the two-block inputs in an affine space $V = U + q$ that we specify at the input of χ of the first round, with U a vector space and q an offset. U is the vector space that spans all the bits in the columns of the two blocks where $a_1 = \lambda^{-1}(b_1)$ is active, and all the other (passive columns) bits are fixed to 0. The number of active columns in the two active blocks is $\mathrm{w}(a_1)$ and hence the dimension of V is $3\mathrm{w}(a_1)$. Knowing the basis of U at the input of χ it is straightforward to obtain the basis at the input of λ by simply applying λ^{-1} to the basis vectors.

Note that, when applied to Kravatte, we do the same but with rows taking the place of columns. In Kravatte there is no one-to-one correspondence between the (minimum reverse) weight of a_1 and the dimension of U, except that it is at most 5 times the minimum reverse weight of a_1.

Since V takes all the possible values in each active column and χ is bijective, the set T obtained by applying χ to the elements of V is an affine space with the same basis as U and it can only differ from V in its offset. The point is now that T contains $|T|/2 = |V|/2$ pairs with difference a_1 in both blocks, that we denote as $a_1||a_1$. We construct these pairs as follows. For each element $u \in T$, construct $u^* = u + (a_1||a_1)$. As $(a_1||a_1) \in U$, we have $u^* \in T$. This gives a pair with difference $a_1||a_1$. We can do this for any element $u \in T$ leading to a total of $|T|/2$ pairs. Note that the attacker cannot identify these pairs a priori due to the presence of the secret masks, but their mere presence in T contributes to the probability of a collision that is easily identifiable a posteriori.

So, we apply $|V|$ messages and have $|V|/2$ pairs at the input of χ of the second round with a difference b_1 in both active blocks. We have basically linearized χ of the first round by applying inputs forming a column-aligned affine space. Clearly, the first difference b_0 vanishes from the equation as does γ and our attack is effectively based on the trail core $(b_1, b_2, b_3, b_4, b_5)$. When applying $M/2$ two-block inputs with $M/2$ a multiple of $|V|$, the success probability now becomes:

$$\Pr(\text{collision}) \approx M2^{-(2+2\mathrm{w}(b_1)+2\mathrm{w}(b_2)+2\mathrm{w}(b_3)+2\mathrm{w}(b_4)+\mathrm{w}(b_5))} .$$

With the current trail weight bounds, found in Table 7, we would obtain an upper bound of $\Pr(\text{collision}) \approx M2^{-(2+54+56)} = M2^{-112}$, higher than the term in our security claim. However, our 4- and 5-round bounds for Xoodoo are not tight as they are just the side effect of the absence of trails with weight below 54 and 56 respectively. Improving the bounds to weights of, say, 15 per round, would be enough to decrease this term well below $M2^{-128}$. Our current bounds on 3 and 6 rounds suggest this can be done by adapting our trail scanning software to 4- and 5-round trails and we consider this future work.

We also consider following more theoretical questions as interesting research problems:

- Is it possible to linearize (some active columns of) the non-linear layer χ of the second round and, if so, what is the cost in terms of data complexity?
- How does success probability behave for values of M that are not large enough to form an affine space that covers all active columns of a_1?
- Can one increase the success probability by constructing structures with multiple input differences that are horizontal shifts of each other?

6.3.3 Properties of mask derivation

The purpose of the mask derivation is to derive the 384-bit mask k from a variable-size key K. To counter attacks that swap input blocks i and $i + \delta$, the adversary should have no effective way to predict the value of $k + \text{roll}_{\text{Xc}}^{\delta}(k)$ by guessing part of the mask k or key K. Regarding collision attacks as described in the previous section, it shall be hard for the adversary to reduce the required dimension of the vector space U by n after guessing less than n bits of the masks (or linear combinations thereof) of the two active blocks.

We have addressed these requirements by having roll_{Xc} operate on the full state and having the non-linear layer χ after the diffusion mapping θ in the round function.

6.3.4 Attacks solely based on outputs

Clearly, two or more blocks of output give enough information to determine the value of the output mask k' and the rolling state, independently of the compression phase or the input that was applied. Extracting it however should be computationally difficult. When performing an algebraic attack using two or more output blocks, the adversary must solve a system of equations with unknown variables spread over two full instances of Xoodoo[6]. The best reference on attacks on the expansion phase is Chaigneau et al. [CFG+18] that discusses attacks on a preliminary version of Kravatte that we will denote as Kravatte'. They used the following techniques:

Meet-in-the-middle They express bits of the intermediate state after q rounds of p_{e} as polynomials of bits of the rolling state $\text{roll}_{\text{e}}^{j}(y)$ on the one hand and as polynomials of the output mask k' on the other, using the knowledge of an output block z_j. The number of monomials in y is limited by the algebraic degree of q rounds of p_{e}, and the number of monomials in k' is limited by the algebraic degree of $n - q$ inverse rounds of p_{e}. As the inverse of χ in Xoodoo has only algebraic degree 2, this technique would likely work better in Xoofff than in Kravatte'.

Linearization They convert non-linear equations to a system of linear equations by considering the monomials as independent variables, so-called monomial variables.

Elimination of monomials by exploiting linear recurrence If roll_{e} is linear, the bits of $\text{roll}_{\text{e}}^{j}(y)$ satisfy a linear recurrence equation. This allows eliminating the monomial variables in $\text{roll}_{\text{e}}^{j}(y)$ from the system of linear equations above, leaving only monomial variables in k'.

If roll_{Xe} would be linear, the attacks of [CFG+18] would probably work much better on Xoofff than on Kravatte'. In Xoofff the inverse of χ has lower degree than in Kravatte' and it has a smaller state. However, roll_{Xe} is not linear and it has been designed with these attacks in mind.

6.3.5 Attacks using input-output pairs

In [BDH+17, Section 5.4] an attack is described that exploits the outputs of a large set of inputs that result in an affine space in the accumulator. In a way it *skips* the application of p_{c} and roll_{c}, by restricting the value of the input blocks in each position to two values. If the dimension D of this affine space is equal to the degree d of $p_{\text{e}} \circ p_{\text{d}}$, the sum of the outputs is independent of the input mask. If $D > d$ this sum is zero and if it $D = d - 1$, each bit of the output sum is a linear function of the mask. The former two can lead to a distinguishing attacks, the latter to key recovery. These attacks impose a lower bound to the degree of $p_{\text{e}} \circ p_{\text{d}}$. In Xoofff these are 12 rounds of Xoodoo and hence the degree approaches the maximum value 383.

In [CFG+18] this attack was improved by guessing output mask bits and peeling off 2 rounds at the end. Still, with 12 rounds of Xoodoo even peeling off 4 rounds would still require applying a set of 2^{256} chosen inputs.

All remaining attacks on Xoofff require some distinguisher in an Even-Mansour like structure, where the input and output masks serve as secret keys and the permutation consists of 18 rounds of Xoodoo. This is the realm of the attacks based on classical distinguishers such as differential and linear cryptanalysis, truncated differentials, impossible differentials, boomerang and rectangle attacks, integral cryptanalysis, and of course invariant subspace and nonlinear invariant attacks. The challenge for the majority of above attacks is that $18 - \epsilon$ rounds need to be bridged with some distinguisher. In the light of the fact that Xoodoo reaches SAC after 3.5 rounds, that Xoodoo has weak alignment and that for 4 rounds or more low-weight differential and linear trails are nowhere to be seen, finding such a distinguisher would be a major breakthrough. Moreover, note that an attacker has only access to the *forward cipher*, as Xoofff, or Farfalle in general, simply has no inverse.

7 Trail analysis

In this section, we prove lower bounds on the weight of differential and linear trails using a computer-aided approach. We base ourselves on the techniques presented by Mella et al. in [MDA17]. The results are in Section 5.2.4.

7.1 Unifying differential and linear trail search

Given the strong similarity between the study of differential and linear trails, we further unify the notation by defining:

- $\lambda^\star = \lambda$, $\rho_{early} = \rho_{\text{east}}$, $\theta^\star = \theta$ and $\rho_{late} = \rho_{\text{west}}$ for differential trails, and
- $\lambda^\star = \lambda^\top$, $\rho_{early} = \rho_{\text{west}}^{-1}$, $\theta^\star = \theta^\top$ and $\rho_{late} = \rho_{\text{east}}^{-1}$ for linear trails.

This is illustrated in Figure 6.

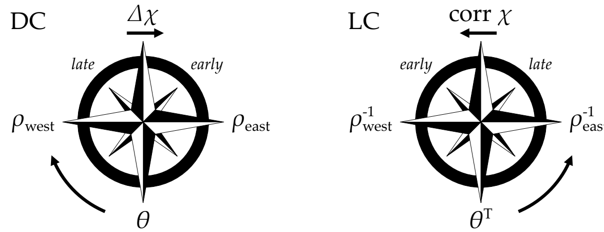

Figure 6: Conventions for differential (DC) and linear (LC) trails in the round function.

7.2 General strategy

Thanks to the similarity of our permutation with Keccak-p, we base our approach on that of Mella et al. in [MDA17]. We exhaustively scan the space of 3-round trail cores and use that to prove lower bounds on the weight of trail cores up to 6 rounds. The 3-round trails are in turn obtained by extending 2-round trails forward and backward. We set our parameters such that all trails up to weight $T_3 = 50$ are generated. Finally, we extend these trails to 6 rounds with the guarantee that any trail with weight $\leq 2T_3 + 2 = 102$ will be found, if it exists.

In a nutshell, the underlying ideas are the following. The weight of a 3-round trail is $\mathrm{w}(b_0) + \mathrm{w}(b_1) + \mathrm{w}(b_2)$. A naive way to generate all trails up to some weight $6n$ would be to generate all patterns b with weight below $2n$ and then extend forward and backward to 3-round trails. The number of such patterns however grows very fast with the weight, i.e., there are $\binom{128}{n}7^n$ patterns with weight $2n$. E.g., for $2n = 10$, this is already about 2^{42}.

The number of patterns can be drastically reduced by considering symmetry. As both χ and λ are invariant with respect to translations parallel to the planes, the number of patterns reduces roughly by a factor 128. But it still grows very fast with the weight.

The weight of a 3-round trail can be expressed alternatively as $\mathrm{w}(a_1) + \mathrm{w}(\lambda(a_1)) + \mathrm{w}(b_2)$ or as $\mathrm{w}(a_1) + \mathrm{w}(a_2) + \mathrm{w}(\lambda(a_2))$. In the former case, two of the three weights are fully determined by a_1 and in the latter case by a_2. In both cases, the sum of those two weights is of the form $\mathrm{w}(a) + \mathrm{w}(\lambda(a))$.

As demonstrated in [MDA17] for Keccak-p, we expect for Xoodoo that the number of trails with a given weight per round decreases with the number of rounds. In other words, we expect that the number of 2-round trails with weight below $4n$ is smaller than that of 1-round trails with weight below $2n$. We will hence scan the space of patterns a taking into account the sum $\mathrm{w}(a) + \mathrm{w}(\lambda(a))$ and extend them forward and backward, thereby dramatically reducing the number of patterns to extend.

Let us detail our approach. A 3-round trail core $Q = a_1 \xrightarrow{\lambda^\star} b_1 \xrightarrow{\chi} a_2 \xrightarrow{\lambda^\star} b_2$ has weight $\mathrm{w}(Q) = \mathrm{w}(a_1) + \mathrm{w}(b_1) + \mathrm{w}(b_2) = \mathrm{w}(a_1) + \mathrm{w}(a_2) + \mathrm{w}(b_2)$. We are looking for all trail cores such that $\mathrm{w}(Q) \leq T_3$. Among these, either $\mathrm{w}(a_1) \leq \mathrm{w}(b_2) + \delta$ or $\mathrm{w}(a_1) > \mathrm{w}(b_2) + \delta$, for some fixed integer δ that we set to $\delta = 2$ in our implementation.

- The former case implies that $2\mathrm{w}(a_1) + \mathrm{w}(b_1) \leq T_3 + \delta$. Such trails can be obtained by generating all 2-round trails $a_1 \xrightarrow{\lambda^\star} b_1$ satisfying this inequality and then extending each of them forward by finding all states a_2 compatible with b_1.
- The latter case implies that $\mathrm{w}(a_2) + 2\mathrm{w}(b_2) < T_3 - \delta$, and since all weights are even, the condition is equivalent to $\mathrm{w}(a_2) + 2\mathrm{w}(b_2) \leq T_3 - \delta - 2$. Such trails can be obtained by generating all 2-round trails $a_2 \xrightarrow{\lambda^\star} b_2$ satisfying this inequality and then extending each of them backward by finding all states b_1 compatible with a_2.

Note that there are two differences between our general approach and that with the one of Mella et al. First, we do not distinguish between kernel and non-kernel states when generating 2-round trail cores. Second, we allow the generation of 2-round trail cores to be unbalanced between those that are extended forward and backward by allowing $\delta \neq 0$. We noticed that in our implementation the backward extension takes more time than the forward extension, so by setting $\delta > 0$ we reduce the number of 2-round trails to extend backward, at the cost of an increase of those to be extended forward, and we reach a more balanced timing in both cases.

7.3 Generation of 2-round trail cores

We now concentrate on the generation of 2-round trail cores of the form $a \xrightarrow{\lambda^\star} b$. We do this by generating state values A at the input of $\theta^\star$, so that we can control the parity of A and exploit the properties of θ. From A, we can compute $a = \rho_{early}^{-1}(A)$ and $b = \rho_{late}(\theta^\star(A))$. We do this while bounding the cost function $\alpha\mathrm{w}(a) + \beta\mathrm{w}(b)$ for $\alpha, \beta \in \{1, 2\}$ as explained above.

7.3.1 Properties of θ

As θ is a linear layer similar to Keccak's θ function, the following definitions are adapted from those in [BDPA11b]. Note that, as a linear function, the properties of θ are the same whether applied on a state absolute value or on a difference, so we just write âĂIJvalueâĂİ.

The *parity plane* (or *parity* for short) $P(A)$ of a value A is defined as the parity of the columns of A, namely $P(A) = \sum_y A_y$. A column is *even* (resp. *odd*) if its parity is 0 (resp. 1). When the parity of a value is zero (i.e., all its columns are even), we say it is in the *column-parity kernel* (or *kernel* for short).

The θ-effect of a value A is $E(A) = P(A) \lll (1,5) + P(A) \lll (1,14)$. A column of coordinates (x,z) is *affected* iff the corresponding bit in $E(A)$ is 1; otherwise, it is *unaffected*. Note that the θ-effect always has an even Hamming weight so the number of affected columns is even. We define the *θ-gap* as the number of affected columns divided by two.

An odd column at coordinates (x,z) induces two affected columns at coordinates $(x+1,z+5)$ and $(x+1,z+14)$. Adding a second odd column at coordinates $(x,z+9)$ will induce an affected column at $(x+1,z+23)$ and cancel the affected column at $(x+1,z+14)$. This can be further extended to more odd columns at coordinates $(x,z+9n)$. Informally, chaining such odd columns, together with their two induced affected columns, makes up a *run*. When all the columns in a sheet are odd, then the θ-effect cancels and there are no induced affected columns. Informally, we call the set of such odd columns a *loop*.

To better formalize this, we can make a change of coordinates on the z-axis and use the t coordinates instead:

$$z = 9t \ \Leftrightarrow t = 25z.$$

An odd column at coordinates (x,t) induces two affected columns at coordinates $(x,t-3)$ and $(x,t-2)$. A run is thus a sequence of odd columns with the same x coordinate and consecutive t coordinates.

7.3.2 Decomposition of a state value around θ

Given a state value A, a bit at position (x,y,z) (or, equivalently, at (x,y,t)) is said to be *active* if its value is 1. Otherwise, it is *passive*.

We decompose a state value at the input of θ as the sum of basis vectors, called *elements*, of three different kinds: the parity loops, the parity runs and the orbitals.

Definition 3. A *parity loop* (or *loop* for short) is a state value with 32 active bits in a sheet, each in a distinct column.

Definition 4. A *parity run* (or *run* for short) is a state value composed of $1 \le l \le 31$ active bits in l different columns of a sheet x with consecutive t coordinates $(t_0, t_0+1, \ldots, t_0+l-1)$, and of zero or two active bits in each of the (affected) columns $(x+1, t_0-3)$ and $(x+1, t_0+l-3)$.

Definition 5. An *orbital* is a state value with two active bits in the same column.

Loops and runs generate odd columns, while runs also have affected columns. From the decomposition of the value before θ, the value at the output of θ is easy to determine: A loop and an orbital are invariant through θ, and a run gets the bits in its two affected columns complemented through θ, while the remaining columns remain unchanged. Since θ is linear, the state after θ can be decomposed in the images of the elements through θ, with the same coefficients. So the decomposition into elements tells as much about the state before as the state after θ.

Any state value can be expressed as the bitwise sum of elements and, as such, they generate the full state space. However, this decomposition is not unique. To make the decomposition unique, we rely on a number of conventions. These conventions also help bounding the weight of the states obtained when combining these elements by avoiding as much as possible turning an active bit back to passive on either side of θ. The conventions are as follows. First, all odd columns stem from a unique loop or run. Then, an orbital can only be added to an empty column or to an unaffected odd column with a single active bit at $y = 0$. Finally, an affected odd column must follow the odd-0 convention:

Figure 7: The odd-0 convention, illustrated. Here, each rectangle represents a column and each circle a bit. A bit can be either passive (white) or active (black). In an affected column, the left half shows the value before $\theta^\star$, while the right half shows the value after $\theta^\star$. The position $y = 0$ is at the bottom.

Definition 6. The *odd-0 convention* says that an affected odd column must be represented as the sum of an unaffected odd column with a single active bit at $y = 0$ (of a loop or of a run) and of an affected even column (of a run) chosen accordingly. Figure 7 illustrates this.

Lemma 1. *The value A at the input of θ can be uniquely decomposed as a sum of elements.*

Proof. Let us define an algorithm that determines the last element of the state, then removes it by adding it back to A. The algorithm can then be applied recursively until all bits are passive.

The algorithm takes as input a value A at the input of θ, computes the parity $P(A)$ and the θ-effect $E(A)$ and then proceeds as follows.

1. First, it looks for the unaffected column with two or more active bits with the highest coordinates using $[x, z]$ lexicographical ordering. If it exists, the algorithm outputs an orbital O by taking the two bits with the highest y coordinates in that column, adds back O to A and recursively starts again.

2. Otherwise, it looks for the odd column with coordinates (x, t) such that $(x, t - 1)$ is even and with the highest coordinates using $[x, z]$ lexicographical ordering. If it exists, the algorithm counts how many consecutive columns are odd, until $(x, t + l)$ is even. It then builds a run R by taking from A the l active bits in each of the l columns from (x, t) to $(x, t + l - 1)$, plus some bits in the affected columns at $(x + 1, t - 3)$ and $(x + 1, t + l - 3)$:

   (a) For each unaffected odd column, the run takes the unique odd active bit.

   (b) For each affected odd column, it follows the odd-0 convention, i.e., the run is defined with an active bit at $y = 0$, independently of what bit is actually active in A.

   (c) If the column at $(x + 1, t - 3)$ is affected even, the run takes the zero or two active bits in that column.

   (d) If the column at $(x + 1, t - 3)$ is affected odd, it follows the odd-0 convention, i.e., the run takes the zero or two active bits such that flipping those bits leaves a single bit at $y = 0$ in that column.

   (e) The last two steps are repeated for the column at $(x + 1, t + l - 3)$.

   It outputs the obtained run R, adds it back to A and recursively starts again.

3. Otherwise, it looks for the sheet with 32 odd columns with the highest x coordinate. If it exists, the algorithm outputs a loop L by taking from A the 32 active bits in each of the 32 columns, adds back L to A and recursively starts again.

4. Otherwise, it returns *none*.

The algorithm is deterministic, so the output is uniquely determined by the value of A. It remains to show that the algorithm always terminates, and that it stops iff $A = 0$.

When step 1 outputs an orbital, it strictly decreases the number of active bits. If step 1 does not output an orbital, it means that all the unaffected columns have either zero or one active bit.

When step 2 outputs a run, it strictly decreases the number of odd columns (by $l \geq 1$) and of affected columns (by 2). Furthermore, the affected columns that are removed leave zero or one active bit behind, thanks to the odd-0 convention. If step 2 does not output a run, it means that there are no more affected columns and that all columns contain at most one active bit. In such conditions, a state is either passive or contains only loops.

When step 3 outputs a loop, it strictly decreases the number of odd columns (by 32). If step 3 does not output a loop, it must be passive so $A = 0$. □

7.3.3 Breaking down to bit-level units

To construct differential and linear trails, we use the tree traversal technique as defined by Mella et al. in [MDA17, Section 3]. We represent a state value as a set of *units*, each consisting of a pattern of active bits. By defining an *ordering* of units, a set of units becomes a *unit list*. A unit list can be constructed progressively by appending a unit at the end. Finally, we need to define a *cost function* and a *subtree bounding function* to define the set of state values we are interested in and to be able to prune the search.

Compared to the choice of more macroscopic units by Mella et al. for the bounds in Keccak [MDA17], we decided to define units that activate at most one bit before θ and at most one bit after θ. More specifically, a unit represents an active bit both before and after θ in unaffected columns, or a bit that is active either before or after θ in affected columns. This allows a finer-grained bounding function, where the decision to set a bit can every time lead to pruning the tree and thus potentially save processing time. We thus break down loops, runs and orbitals in a number of bits, called *bit units*.

A bit unit is first characterized by the type of the element it composes, and that can be a loop, a run or an orbital. The ordering is primarily on the type and such that loops come first, then runs and finally orbitals:

$$\text{loop} \prec \text{run} \prec \text{orbital}.$$

- A loop-typed bit unit represents an active bit in a loop and is characterized by its (x, y, z) coordinates. After the type, the ordering is lexicographic on $[x, z, y]$. Since a loop is bound to a given sheet x, all the bit units composing a loop are consecutive in the unit list.

- A run-typed bit unit represents an active bit in an odd column or in an affected column of a run. It is first characterized by the (x_0, z_0) (or, equivalently, (x_0, t_0)) coordinates of the first odd column. We then distinguish the different bit units by their rank r and subrank s, and either by their y coordinate or their value v. After the type, the ordering is lexicographic on $[x_0, z_0, r, s, y, v]$ so that the bit units composing the same run are consecutive. The different bit units are as follows, in this order:

  – For the affected column at $(x_0 + 1, t_0 - 3)$, there are three bit units with rank $r = 0$ and subrank $s \in \{-3, -2, -1\}$. Since these units represent what happens in an affected column, they are also characterized by a value $v \in \{\text{before}, \text{after}\}$ telling whether the bit is active before or after θ. The effective position of the active bit is $(x_0 + 1, s + 3, t_0 - 3)$.

- For each odd column, there is bit unit representing an active bit at $(x_0, y, t_0 + r)$ for rank $r \in \{0, 1, \ldots, l-1\}$, subrank $s = 0$ and y coordinate. These bits are active before and after θ.
- For the affected column at $(x_0 + 1, t_0 + l - 3)$, there are again three bit units characterized by their value v and, this time, with rank $r = l - 1$ and subrank $s \in \{1, 2, 3\}$. The effective position of the active bit is $(x_0 + 1, s - 1, t_0 + r - 2)$.

- An orbital-typed bit unit represents an active bit in an orbital and is characterized by its (x, y, z) coordinates. After the type, the ordering is lexicographic on $[x, z, y]$ so that the bit units composing an orbital are consecutive.

As already said, the ordering of units is defined such that the units that compose an element are consecutive in the unit list. Additionally, the ordering is chosen in agreement with Lemma 1. That is, for a state value represented as a unit list, the last units correspond to the element returned by the algorithm in the proof of Lemma 1.

7.3.4 Lower bounding the cost

Recall that we are interested in the cost function $\alpha\mathrm{w}(\rho_{early}^{-1}(A)) + \beta\mathrm{w}(\rho_{late}(\theta^\star(A)))$ for $\alpha, \beta \in \{1, 2\}$ as explained above.

In a unit list, we distinguish between *stable* and *unstable bits*. A stable bit is an active bit that is guaranteed to stay active even if more bit units are added to the unit list. We characterize stable bits as follows.

- If the unit list does not contain orbital-typed bit units, an active bit in a loop or in a run is stable iff its y coordinate is not zero.
- If the unit list contains at least one orbital-typed bit unit, all active bits are stable.

The rationale is as follows. If an active bit in a loop or in a run is added to an unaffected column, it is active both before and after θ. But if the column it sits in becomes affected due to the addition of a run, the active bit is removed from either before or after θ. Similarly, a bit in an even affected column is active at a given side of θ, but a new active bit from a loop or a run can be added to it, effectively changing the side where the old bit is active. However, the odd-0 convention prevents an affected column from being added to an odd column where the active bit is at $y > 0$ and, vice-versa, it prevents an active bit of a loop or a run with $y > 0$ from being added to an affected column. So once an active bit with $y > 0$ is added, it cannot be removed. Furthermore, an orbital is stable and comes after other types of elements, so once we start adding orbital-type bits, all the bits are stable.

The subtree bounding function starts by counting the contribution of the stable bits after translations through ρ_{early}^{-1} and ρ_{late}. An active bit contributes the weight by 2 only if it lands in a column without any active bits yet. Then the subtree bounding function lower-bounds the contribution of the unstable bits. Notice that an unstable bit will yield an active bit at least in either side of θ. So the subtree bounding function counts the minimum of the contribution of an active bit before or after θ. It also marks the column where the unstable bit lands, on both side of θ, so that the contribution in a given column cannot be counted more than once.

7.4 Extension to 3 and 6 rounds

For every 2-round trail core produced, we extend it forward or backward according to the general strategy outlined in Section 7.2 above.

The extension exploits the fact that the compatible states form an affine space, as shown in Corollary 1. Let us illustrate this for the forward extension, as the backward

extension enjoys the same property and the description can be easily adapted to that case. For each active column at the input of χ, we form an affine space of the compatible states at the output of χ. We do this for all active columns, resulting in a description of an affine space at the output of χ: $O + \langle B_1, B_2, \ldots, B_w \rangle$. We then transform the offset and basis vectors through $\lambda^\star$ to get a description of the affine space at the input of the next χ, namely $O' + \langle B'_1, B'_2, \ldots, B'_w \rangle$ with $O' = \lambda^\star(O)$ and $B'_i = \lambda^\star(B_i)$. This way, we can more easily compute (and bound) the weight that is added to the trail when extending it.

To help bound the weight of the trail when extending it, we first triangularize the basis vectors (B_i). The triangularization defines a nested sequence of sets $(\mathcal{B}_i)$ of bit positions such that $i < j \Rightarrow \mathcal{B}_i \supset \mathcal{B}_j$ and B_i has no bits 1 outside of $\mathcal{B}_i$. The search through the affine space also uses the technique of tree search of Mella et al. [MDA17]. Here, the units represent the basis vectors and their ordering is according to their indexes, i.e., $i < j \Leftrightarrow B_i \prec B_j$. This way, if the current unit list ends with B_i, we can lower bound the weight of all its descendants given that all bits outside of $\mathcal{B}_{i+1}$ cannot be changed.

A 6-round trail core Q as

$$Q = a_1 \xrightarrow{\lambda^\star} b_1 \xrightarrow{\chi} a_2 \xrightarrow{\lambda^\star} b_2 \xrightarrow{\chi} a_3 \xrightarrow{\lambda^\star} b_3 \xrightarrow{\chi} a_4 \xrightarrow{\lambda^\star} b_4 \xrightarrow{\chi} a_5 \xrightarrow{\lambda^\star} b_5$$

has a weight of $\mathrm{w}(Q) = \mathrm{w}(a_1) + \sum_{i=1}^{5} \mathrm{w}(b_i)$. If $\mathrm{w}(Q) \leq 2T_3 + 2 = 102$, then $\mathrm{w}(a_1) + \mathrm{w}(b_1) + \mathrm{w}(b_2) \leq T_3$ or $\mathrm{w}(b_3) + \mathrm{w}(b_4) + \mathrm{w}(b_5) = \mathrm{w}(a_4) + \mathrm{w}(b_4) + \mathrm{w}(b_5) \leq T_3$. So if such a trail exists, we must find it when extending all 3-round trails of weight at most T_3 both backward and forward.

7.5 Concrete experiments

We performed the generation of 2-round trail cores and their extension to 3 rounds together. The search for both differential and linear trails of 3 rounds took about 16 core×days on a desktop PC equipped with an Intel® CoreâĎć i5-6500 CPU. We ran the computation on four parts in parallel, split between linear and differential trails and between the forward and backward extensions, with the longest part taking 5 days. The extension to 6 rounds took a few minutes.

The source code of the program to generate and extend trails is available as open source software [DHAK18b]. We wrote it in C++ and used parts of KeccakTools [BDP+17], in particular for the generic tree search code from Mella at al. [MDA17]. We improved the generic tree search code slightly, then instantiated it with the appropriate classes for the generation of 2-round trail cores. The resulting trail cores listed in Table 6 are also given, with differential and linear trails split in different files [DHAK18b]. Each set of trails is provided both in a format easily parsable by the software and in a visual text representation.

7.6 Inherent 3-round trails

We now describe trails that are inherent to the very structure of Xoodoo.

Definition 7. We say that a trail is *inherent* if a trail with the same structure exists for any variant of Xoodoo with θ that is a column parity mixer, with χ that allows one-active-bit patterns to propagate to the same one-active-bit pattern, and with ρ_{west} and ρ_{east} that consist of plane shifts.

For convenience, we restrict to the case where one odd column in θ makes 2 columns affected, but this could be easily generalized to another number of affected columns, and the number of active bits in the sequel would need to be adapted. Table 9 shows the three types of inherent trails with weight up to 38 and Figure 8 schematically depicts them.

Table 9: Inherent trails

name	θ-gap profile	weight profile	#
Vortex	$\|0\|0\|$	$12 + 12 + 12 = 36$	1
Single-orbital fan	$\|0\|2\|$	$4 + 4 + 28 = 36$	3
θ^2-glide	$\|1\|2\|$	$10 + 6 + 22 = 38$	3

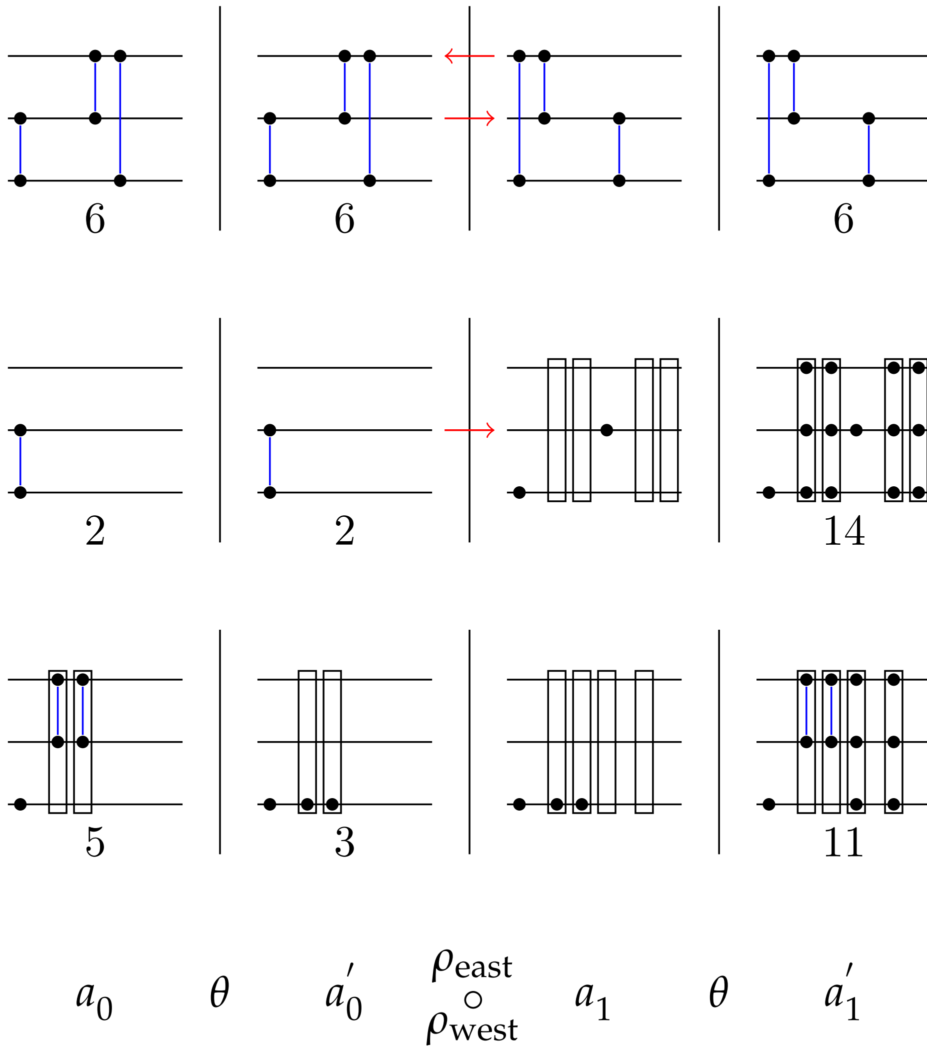

Figure 8: Schematics of three types of inherent trails: vortex (top), single-orbital fan (middle) and θ^2 glide (bottom).

In all three types of inherent trail, the nonlinear layer χ of the middle round acts as the identity. This means that we can replace $\rho_{early} \circ \chi \circ \rho_{late}$ with $\rho_{early} \circ \rho_{late}$, and we denote it by ρ_{both}. Moreover, in the nominal case, all active bits in the three χ layers appear in different columns, hence each making one column active. So, we study trails (a_0, a'_0, a_1, a'_1) that propagate through $\theta^\star \circ \rho_{both} \circ \theta^\star$ with $a'_0 = \theta^\star(a_0))$, $a_1 = \rho_{both}(a'_0)$ and $a'_1 = \theta^\star(a_1)$:

$$Q = a_0 \xrightarrow{\theta^\star} a'_0 \xrightarrow{\rho_{both}} a_1 \xrightarrow{\theta^\star} a'_1 \ .$$

The trail weight equals the sum of the Hamming weights of a_0, a'_0 (or equivalently a_1 as ρ_{both} is a mere bit transposition) and a'_1 times 2. Note that the value of a_0 determines the full trail.

In the first inherent trail the input to θ in both rounds is in the kernel making it vanish. In such trails, a'_0 is called a *vortex* in [DA12] and it is a state that is in the kernel and remains so after applying ρ_{both}. Clearly, as all patterns in the trail have the same Hamming weight, the trail weight is 6 times the Hamming weight of a_0. Vortices are completely determined by ρ_{both}, and more specifically, by its (two-dimensional) translation offsets for the planes 1 and 2. Let us denote these by u_1 and u_2. Due to the fact that ρ_{both} treats the three planes as rigid structures, there exists a vortex consisting of 3 orbitals, that ρ_{both} maps to 3 orbitals. In particular, a'_0 and a_1 have active bits in in the following positions:

- Plane 0: in 0 and $u_1 - u_2$ before ρ_{both}, then in 0 and $u_1 - u_2$ after ρ_{both}
- Plane 1: in 0 and $-u_2$ before ρ_{both}, then in u_1 and $u_1 - u_2$ after ρ_{both}
- Plane 2: in $-u_2$ and $u_1 - u_2$ before ρ_{both}, then in 0 and u_1 after ρ_{both}

Note that the existence of such a 3-orbital vortex is independent of the dimension of the rigid structures (in our case planes) and so is the weight of the corresponding 3-round trails: 36.

A *single-orbital fan* is a trail that has two active bits in the same column in a_0. Since a_0 is in the kernel, $a'_0 = a_0$. These bits propagate through ρ_{both}, where they land in different columns before $\theta^\star$. They then expand to 7 bits each after $\theta^\star$ in a'_1, and these 2×7 bits can land in 14 different columns. Since a_0 and a'_0 contain 2 active bits and a'_1 contain 14 active bits, the total weight is thus 36.

A θ^2*-glide* is a trail with the following structure:

- In a_0, the state is made of one active bit alone in a column and of two orbitals located in the affected columns induced by the first active bit.
- In a'_0, the two orbitals are replaced by a single active bit each, in addition to the first active bit that remains. We choose the positions of the bits in a_0 such that they all arrive in the same plane in a'_0.
- Since all the active bits are in the same plane, ρ_{both} acts as a global shift of the whole state, which we can ignore in this discussion and consider that $a_1 = a'_0$.
- Finally, $a'_1 = \theta^\star(a_1) = \theta^\star(\theta^\star(a_0))$ up to the global shift. The $(\theta^\star)^2$ operation applied to the single active bit of a_0 induces 4 affected columns, two going from 1 to 2 active bits, and two going from 0 to 3 active bits, so with a total of $1 + 4 + 6 = 11$ active bits.

Since a_0 contains 5 active bits, a'_0 contains 3 active bits and a'_1 contains 11 active bits, the total weight is thus 38.

8 Conclusions and perspectives

In this paper, we have introduced a novel permutation called Xoodoo and a deck function called Xoofff for concrete encryption and authentication applications.

From a cryptographic point of view, we think that the chosen structure and set of operations lead to a design with nice properties. It is easier to analyze the differential and linear trail propagation than on Keccak, and we could make sure that the chosen rotation constants avoid low-weight trails that are not inherent to the structure. Our permutation has a dispersion layer between every mixing and nonlinear layer, whereas in Keccak-p the mixing layer follows the nonlinear layer immediately, causing suboptimal intra-slice effects. This may explain why the minimum weight over 3-round trails is 36 for Xoodoo, while it is 24 for differential trails in Keccak-p[400] [MDA17]. Furthermore, there is no known bounding of linear trails in Keccak-p[400].

From an implementation point of view, we expect that Xoodoo shares with Keccak-p highly efficient hardware implementations (and much smaller than Keccak-p[1600]) and efficient protections against side-channel attacks. Looking back at Table 5, the performance of Xoofff on Skylake(X) processors is excellent, competing with that of the AES in counter mode although the latter benefits from hardware acceleration and the former uses only general-purpose instructions. On 32-bit processors, Xoofff is much faster than the AES in counter mode. As for the performance on ARM Cortex processors as reported in Table 4, we notice that the performance per round of Xoodoo on ARM Cortex-M3 is similar to that of Gimli, while Gimli is significantly faster on Cortex-M0. However, these values have to be taken with care, as each permutation does not need the same number of rounds when used in a given mode to yield a secure scheme. For instance, Xoodoo needs 6 rounds to guarantee the absence of trails with weight less than 104, while Gimli would need 16 rounds as suggested by [BKL+17, Table 1].

For future work, defining Xoodoo variants with other dimensions can be useful. For constrained platforms, it could be interesting to have permutations with smaller widths. If we take the birthday bound on the permutation width as the limiting factor and we consider the minimum processor word length to be 32 bits, two particular widths come to mind: $288 = 32 \times 3 \times 3$ for targeting 128-bit security and $192 = 32 \times 2 \times 3$ for 80-bit security. This can be realized by taking planes consisting of 3 and 2 lanes respectively. On the other hand, one can also have “planes” consisting of only a single lane of length 96 and 64 respectively. Efficient implementations on 32-bit platform can then be derived by rewriting the operations using bit interleaving [BDP+12]. Towards larger permutations, a 768-bit variant with planes consisting of 4 lanes of 64 bits each would be interesting for sponge-based hashing.

Acknowledgements

Joan Daemen is supported by the European Research Council under the ERC advanced grant agreement under grant ERC-2017-ADG Nr. 788980 ESCADA.

References

[BDH+17] G. Bertoni, J. Daemen, S. Hoffert, M. Peeters, G. Van Assche, and R. Van Keer, *Farfalle: parallel permutation-based cryptography*, IACR Trans. Symmetric Cryptol. **2017** (2017), no. 4, 1–38.

[BDP+12] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, and R. Van Keer, Keccak *implementation overview*, May 2012, https://keccak.team/papers.html.

[BDP+17] ———, KeccakTools *software*, December 2017, https://github.com/KeccakTeam/KeccakTools.

[BDPA11a] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, *On alignment in* Keccak, ECRYPT II Hash Workshop 2011, 2011.

[BDPA11b] ———, *The* Keccak *reference*, January 2011, https://keccak.team/papers.html.

[Ber08a] D. J. Bernstein, *Chacha, a variant of Salsa20*, Workshop Record of SASC 2008, 2008.

[Ber08b] ———, *The Salsa20 family of stream ciphers*, New Stream Cipher Designs - The eSTREAM Finalists (M. Robshaw and O. Billet, eds.), Lecture Notes in Computer Science, vol. 4986, Springer, 2008, pp. 84–97.

[BHMT02] G. Brassard, P. Hoyer, M. Mosca, and A. Tapp, *Quantum amplitude amplification and estimation*, Contemporary Mathematics **305** (2002), 53–74.

[BKL+17] D. J. Bernstein, S. Kölbl, S. Lucks, P. Maat Costa Massolino, F. Mendel, K. Nawaz, T. Schneider, P. Schwabe, F.-X. Standaert, Y. Todo, and B. Viguier, *Gimli : A cross-platform permutation*, Cryptographic Hardware and Embedded Systems - CHES 2017, Proceedings (W. Fischer and N. Homma, eds.), Lecture Notes in Computer Science, vol. 10529, Springer, 2017, pp. 299–320.

[BW99] A. Biryukov and D. A. Wagner, *Slide attacks*, Fast Software Encryption, 6th International Workshop, FSE '99, Rome, Italy, March 24-26, 1999, Proceedings (L. R. Knudsen, ed.), Lecture Notes in Computer Science, vol. 1636, Springer, 1999, pp. 245–259.

[CFG+18] C. Chaigneau, T. Fuhr, H. Gilbert, J. Guo, J. Jean, J.-R. Reinhard, and L. Song, *Key-recovery attacks on full Kravatte*, IACR Trans. Symmetric Cryptol. **2018** (2018), no. 1, 5–28.

[DA12] J. Daemen and G. Van Assche, *Differential propagation analysis of Keccak*, Fast Software Encryption (FSE) 2012. Revised Selected Papers (A. Canteaut, ed.), Lecture Notes in Computer Science, vol. 7549, Springer, 2012, pp. 422–441.

[Dae95] J. Daemen, *Cipher and hash function design strategies based on linear and differential cryptanalysis, PhD thesis*, K.U.Leuven, 1995.

[DHAK18a] J. Daemen, S. Hoffert, G. Van Assche, and R. Van Keer, *Xoodoo cookbook*, IACR Cryptology ePrint Archive **2018** (2018), 767.

[DHAK18b] ———, *XooTools software*, https://github.com/XoodooTeam/Xoodoo, 2018.

[DR02] J. Daemen and V. Rijmen, *The design of Rijndael: AES - the advanced encryption standard*, Information Security and Cryptography, Springer, 2002.

[DR07] ———, *Plateau characteristics*, IET Information Security **1** (2007), no. 1, 11–17.

[DR10] ———, *Refinements of the ALRED construction and MAC security claims*, IET Information Security **4** (2010), no. 3, 149–157.

[DR14] ———, *The MAC function Pelican 2.0*, IACR Cryptology ePrint Archive **2005** (2014), 8.

[GJMN16] R. Granger, P. Jovanovic, B. Mennink, and S. Neves, *Improved masking for tweakable blockciphers with applications to authenticated encryption*, Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I (Marc Fischlin and Jean-Sébastien Coron, eds.), Lecture Notes in Computer Science, vol. 9665, Springer, 2016, pp. 263–293.

[Gro96] L. K. Grover, *A fast quantum mechanical algorithm for database search*, Proceedings of the 28th Annual ACM Symposium on the Theory of Computing, May 1996 (Gary L. Miller, ed.), ACM, 1996, pp. 212–219.

[JNP14] J. Jean, I. Nikolic, and T. Peyrin, *Tweaks and keys for block ciphers: The TWEAKEY framework*, Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014, Proceedings, Part II (P. Sarkar and T. Iwata, eds.), Lecture Notes in Computer Science, vol. 8874, Springer, 2014, pp. 274–288.

[Knu94] L. R. Knudsen, *Truncated and higher order differentials*, Fast Software Encryption, Proceedings (B. Preneel, ed.), Lecture Notes in Computer Science, vol. 1008, Springer, 1994, pp. 196–211.

[LAAZ11] G. Leander, M. A. Abdelraheem, H. AlKhzaimi, and E. Zenner, *A cryptanalysis of PRINTcipher: The invariant subspace attack*, Advances in Cryptology - CRYPTO 2011 (P. Rogaway, ed.), Lecture Notes in Computer Science, vol. 6841, Springer, 2011, pp. 206–221.

[Log] Real Time Logic, *SHARKSSL v2.3.3 crypto library – benchmarks with ARM Cortex-M0*, https://realtimelogic.com/products/sharkssl/Cortex-M0/.

[MDA17] S. Mella, J. Daemen, and G. Van Assche, *New techniques for trail bounds and application to differential trails in Keccak*, IACR Trans. Symmetric Cryptol. **2017** (2017), no. 1, 329–357.

[SD18] K. Stoffelen and J. Daemen, *Column parity mixers*, IACR Trans. Symmetric Cryptol. **2018** (2018), no. 1, 126–159.

[SS] P. Schwabe and K. Stoffelen, *AES on the ARM Cortex-M3 and M4*, Early Symmetric Crypto (ESC), Canach, Luxembourg, January 2017, https://ko.stoffelen.nl/talks/20170119-esc.pdf.

[SYY12] P. Schwabe, B.-Y. Yang, and S.-Y. Yang, *SHA-3 on ARM11 processors*, Progress in Cryptology - AFRICACRYPT 2012 (A. Mitrokotsa and S. Vaudenay, eds.), Lecture Notes in Computer Science, vol. 7374, Springer, 2012, pp. 324–341.

[WT85] A. F. Webster and S. E. Tavares, *On the design of S-boxes*, Advances in Cryptology - CRYPTO '85, Proceedings (H. C. Williams, ed.), Lecture Notes in Computer Science, vol. 218, Springer, 1985, pp. 523–534.

A Constants for any number of rounds

We here detail how the round constants are constructed and, following the formula, how to compute them for any number of rounds.

The round constants C_i are planes with a single non-zero lane at $x = 0$. We specify the value of the lanes at $x = 0$ in the round constants as binary polynomials $p_i(t)$ where the coefficient of t^i denotes the bit of the lane with coordinate $z = i$. We define $p_i(t)$ in terms of a polynomial $q_i(t)$ and an integer s_i in the following way:

$$p_i(t) = t^{s_i}\left(q_i(t) + t^3\right) \text{ with } q_i(t) = t^i \bmod 1 + t + t^3 \text{ and } s_i = 3^i \bmod 7\,.$$

The sequence of polynomials $q_i(t)$ has period 7 and the sequence of offsets s_i has period 6. It follows that the sequence of round constants $\mathrm{C}_i(t)$ have period 42. An instance of Xoodoo with n_r rounds uses the round constants with indices $1 - r$ to 0. We list the round constants with indices -11 to 0 in Table 10.

Table 10: The round constants with indices -11 to 0

i	q_i	s_i	c_i	in hex
-11	$1+t$	3	$t^3+t^4+t^6$	`0x00000058`
-10	$t+t^2$	2	$t^3+t^4+t^5$	`0x00000038`
-9	$1+t+t^2$	6	$t^6+t^7+t^8+t^9$	`0x000003C0`
-8	$1+t^2$	4	$t^4+t^6+t^7$	`0x000000D0`
-7	1	5	t^5+t^8	`0x00000120`
-6	t	1	t^2+t^4	`0x00000014`
-5	t^2	3	t^5+t^6	`0x00000060`
-4	$1+t$	2	$t^2+t^3+t^5$	`0x0000002C`
-3	$t+t^2$	6	$t^7+t^8+t^9$	`0x00000380`
-2	$1+t+t^2$	4	$t^4+t^5+t^6+t^7$	`0x000000F0`
-1	$1+t^2$	5	$t^5+t^7+t^8$	`0x000001A0`
0	1	1	t^1+t^4	`0x00000012`