

BACHELOR THESIS
COMPUTER SCIENCE



RADBOUD UNIVERSITY

Xoodoo Trail Analysis

Author:
Constantin Blach
s4329872

First supervisor/assessor:
Joan Daemen
joan@cs.ru.nl

Second supervisor:
Bart Mennink
b.mennink@cs.ru.nl

August 20, 2019

Abstract

This thesis presents a trail analysis of the 48-byte cryptographic permutation Xoodoo, which was developed by the Keccak team. Xoodoo is a very efficient permutation, which can easily be used on low-end processors. When this permutation was designed, no further analysis on its security has been done, leading to some problems during the permutation for specific cases. These mainly occur due to symmetric states of the Xoodoo permutation, which might leak valuable information when exploited correctly. In this research we created trails, out of the whole trail space, up to a given weight, in order to find the source of the weaknesses. This analysis shows, that there exists better Xoodoo configurations than the one, which is currently used. Based on this, a new Xoodoo-like permutation could be build, which would be more efficient.

Contents

1	Introduction	2
1.1	What is Xoodoo?	2
1.2	Permutation rounds	3
1.3	Making an even better Xoodoo-like permutation	6
2	Preliminaries	7
2.1	What is cryptanalysis?	7
2.1.1	Differential Cryptanalysis	7
2.1.2	Linear Cryptanalysis	8
2.1.3	Weight of a Trail	8
2.2	Xoodoo Trails	9
2.3	Symmetry Properties	12
3	Research	15
3.1	Adjustments on Xoodoo	15
3.2	Trail Analysis Results	15
4	Conclusions	18
A	Appendix	21
A.1	Functions added	21

Chapter 1

Introduction

1.1 What is Xoodoo?

Xoodoo is a 48-byte cryptographic permutation designed by the Keccak team (Bertoni et al.) first presented in 2017 [1]. It was inspired by the previously developed Keccak-p algorithm [6] to be efficient on low-end processors, where the number of registers is limited, but also on high-end processors and in specifically dedicated hardware. The limited number of registers on low-end processors makes the usage of Keccak-p[1600] and Keccak-p[800] inefficient, due to a high amount of swaps in and out of the registers. Xoodoo however is designed to fit in these types of processors because it needs less registers than Keccak-p.

The permutation operates on a 384 bit 3-dimensional array, which has a size of $4 \times 3 \times 32$. This array is the Xoodoo state, which is build by three equally sized planes stacked above each other, where each is of the size 4×32 (i.e. 128 bits). These planes are labelled by the y-coordinate, where $y = 0$ corresponds to the lowest plane and $y = 2$ to the top plane. Additionally, the x-coordinate is used to index the lane and the z-coordinate will describe the bit in each lane (Figure 1.1). Thus, each bit of a Xoodoo state can be referenced by the coordinates (x, y, z) and a specific column by (x, z) . If we want to describe a lane, we can refer to it using the (x, y) coordinates accordingly. One Xoodoo state can be seen as a collection of 128 3-bit columns, which are arranged in a 4×32 array. These columns count as one entry of the array, consisting of three bits, each from one of the three planes. So the value of one column is represented by a 3-bit linear array, for which the x- and z-value are the same. A toy-sized state, similar to a Xoodoo state, which uses only 96 bits in a $4 \times 3 \times 8$ array, can be seen in Figure 1.1. Additionally, different parts of the state are highlighted, such that referring to them later on will be easier.

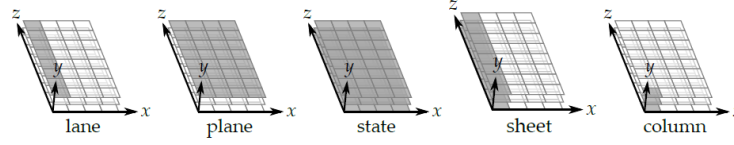


Figure 1.1: Toy-sized Xoodoo version (96 bits) with different highlights, taken from [8]

Xoodoo is no cryptographic function on itself, but can be used to build efficient cryptographic functions. For example, it can be used together with the Farfalle construction [2] in order to create the deck function *Xoofff* [9].

Another, more recent example, of the usage of Xoodoo can be found in Xoodyak, which is a cryptographic primitive used for hashing, encryption, MAC computation and authenticated encryption (AE) [3].

The Xoodoo permutation is designed in a way, such that each bit in the Xoodoo state is dependant on many others, leading to a high diffusion in the state. Xoodoo uses a round function on the state to permute the input, where the number of rounds is variable and will depend on the construction in which Xoodoo is used. For Xoofff, Xoodoo has 6 permutation rounds while for Xoodyak it will even has 12.

During each round, Xoodoo performs a series of rotations, bit-shifts and bitwise additions on the state. These shift constants, which determine the bit-shifts during the rounds, were chosen by the designers of Xoodoo on construction, to hopefully bring the wanted security with them.

1.2 Permutation rounds

As already said, Xoodoo works in rounds which are indicated by R_i , where i denotes the number of the round. Each round consists of 5 steps, where we can differentiate between linear- and non-linear steps. Additionally, four of the five steps of each round are operating on a symmetric basis, which results in the same computation each round, no matter which bits of the states are set. According to *The Design of Xoodoo and Xoofff* [8] the round function consists of the following steps:

1. a mixing layer θ with the shift offsets (1, 5) and (1, 14)
2. a plane shifting step ρ_{west} with the shift offsets (1, 0) and (0, 11)
3. the addition of a round constant ι
4. a non-linear layer χ
5. a second plane shifting ρ_{east} with the shift offset (2, 8)

For the steps θ , ρ_{east} and ρ_{west} a shift offset is also given, which defines the bit shifts. The exact calculations, which are used during the rounds of Xoodoo, can be seen in Algorithm 1, which was taken from [9].

Table 1.1: Notational conventions for Algorithm 1

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

Algorithm 1 Definition of $XOODOO[n_r]$ with $[n_r]$ the number of rounds

Require: Number of rounds n_r

for round index i from $1 - n_r$ **do** $A = R_i(A)$

A Round 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 :

$$A_0 \leftarrow A_0 + C_i$$

χ :

$$B_0 \leftarrow \overline{A_1} * A_2$$

$$B_1 \leftarrow \overline{A_2} * A_0$$

$$B_2 \leftarrow \overline{A_0} * A_1$$

$$A_y \leftarrow A_y + B_y \text{ for } y \in 0, 1, 2$$

ρ_{east} :

$$A_1 \leftarrow A_1 \lll 0, 1$$

$$A_2 \leftarrow A_2 \lll (2, 8)$$

The effect of θ , χ , ρ_{east} , and ρ_{west} on the Xoodoo state, during one permutation round, is illustrated in Figure 1.2, 1.3 and 1.4, which are also taken from [8].

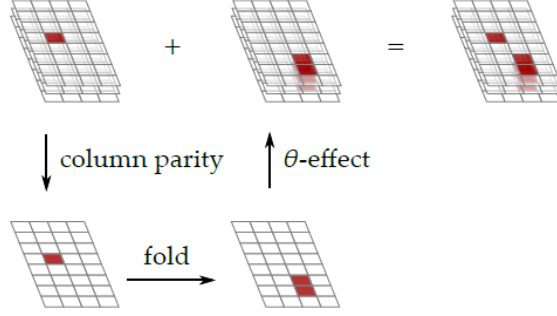


Figure 1.2: Effect of θ on a single-bit state

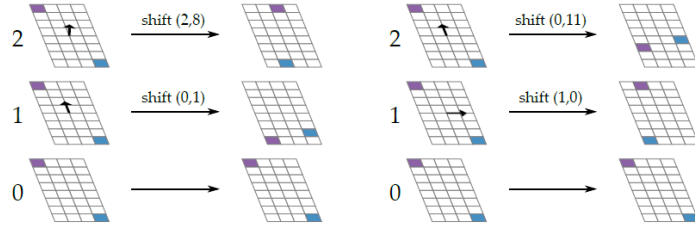


Figure 1.3: Illustration of ρ_{east} (left) and ρ_{west} (right)

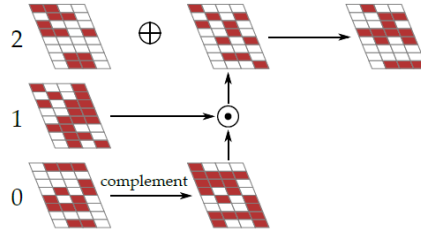


Figure 1.4: Effect of χ on one plane

In the ι step, which is not illustrated above, we add a round constant C_i to the lowest plane of the Xoodoo state, with the y-coordinate $y = 0$. These C_i planes, which also have a size of 4×32 , only have one single non-zero lane at $x = 0$, the rest is set to zero. The round constant may be one of eleven different values, based on the round i we are currently in.

The round function is defined by the sequence of steps $\rho_{east} \circ \chi \circ \iota \circ \rho_{west} \circ \theta$. For this study however we will re-phase the permutation round as it was done in [8]. Instead of starting with θ , each round will start with ρ_{east} and end in

χ . This makes it possible to group the all of the linear mappings together in $\lambda = \rho_{west} \circ \theta \circ \rho_{east}$, which lead to the re-phased round function of $\chi \circ \iota \circ \lambda$.

1.3 Making an even better Xoodoo-like permutation

After Xoodoo was published, it was found that the shift offsets, which are the vectors used during the permutation rounds by θ , ρ_{west} , and ρ_{east} , could have been chosen differently, leading to better propagation properties of the permutation. With other words, these shift offsets produce some very bad patterns when used on 4 rounds of the permutation, which makes it possible for an adversary to gather information about the initially given data. These *low-weight trails* (explained in 2.1.3) can probably be avoided when different shift constants are used. A *trail* consists of a series of states, that show the changes on the input for each step during a round, up until the last round.

To address this problem, a trail analysis has been done to identify weak shift constants and replace them with better ones. This analysis is an exhaustive scan of the trail space of Xoodoo, in which almost all possible combinations of shift constants have been examined. For each combination, a 3-round trail was created, which then was rated by the worst trail found, creating the lower trail bound for that setup, as it was done by Mella et al. in [10].

This research presents new shift constants, which can be used to build a new Xoodoo-like permutation with increased 3-round lower trail bounds. If we use a permutation with better trails, we do not need as many rounds for the same security, than it is currently done. Thus, one possible consequence of these increased lower-trail weights might be a reduction of rounds, keeping the security level but increasing the efficiency.

Chapter 2

Preliminaries

2.1 What is cryptanalysis?

In general, cryptanalysis is used to analyse a given cryptosystem carefully in order to gather information (such as the key or the given input) about the algorithm, which is supposed to be hidden to an adversary. For the cryptanalysis of Xoodoo, we were using a differential- as well as a linear trail analysis approach [10], in order to evaluate the resistance of Xoodoo against attacks that exploit differential propagation and correlation respectively

2.1.1 Differential Cryptanalysis

For differential cryptanalysis (DC) the analyst has to have the possibility to input a plain text P_i to the algorithm in order to receive the corresponding cipher text C_i for that specific input. This is done for many different inputs, where for each input, the resulting cipher text is recorded. Afterwards pairs are created, which consists of two input plain texts P_i and P_j , such that each pair has the same differential value, which is called Δ_{in} , or *input difference* [8]. This value is, for example, calculated by $P_i \oplus P_j$.

Additionally, we create pairs of *output differences*, called Δ_{out} using the same method as for Δ_{in} . This means, we create pairs of (C_i, C_j) , which have the same difference between each other.

Afterwards, we analyse the differential propagation probabilities (DP) of the differentials $(\Delta_{in}, \Delta_{out})$. This is the chance, that an input difference of Δ_{in} will also result in a specific output difference Δ_{out} . To calculate this probability, we count all Δ_{in} and divide them by the number of Δ_{out} :

$$DP(\Delta_{in}, \Delta_{out}) = \frac{\#\Delta_{in}}{\#\Delta_{out}}.$$

This probability indicates how often a differential sequence occur in the state. A higher probability means, that there exists a high amount of pairs

$(\delta_{in}, \delta_{out})$ more often than others, which could be exploited to derive information the permutation or the plain text. This would be an analysis of *differentials*, analysing the dependencies between the in- and output.

For the presented Xoodoo Analysis however, we investigate the DP of *differential trails* rather than only differentials. These n -round differential trails are, as the name already suggests, not only pairs of the two in- and output-values but a sequence of n round differentials (a_i, a_{i+1}) , which are fully specified by the sequence (a_0, a_1, \dots, a_n) [8]. Compared to "only" differentials, as mentioned above, we not only compare the differences between the in- and output of the algorithm but also the differences between each of the permutation rounds. To analyse the trails in more detail, we also include the differences after each linear layer λ , where $b_i = \lambda(a_i)$. With this definition, we can represent each trail by $(a_0, b_0, a_1, b_1, \dots, b_{n-1}, a_n)$. For a full trail Q , we then receive the following representation, as it was done for Keccak in [5]:

$$Q = a_0 \xrightarrow{\lambda} b_0 \xrightarrow{\chi} a_1 \xrightarrow{\lambda} b_1 \xrightarrow{\chi} \dots \xrightarrow{\chi} a_{n-1} \xrightarrow{\lambda} b_{n-1} \xrightarrow{\chi} a_n.$$

Based on the differential propagation probability of a trail Q , we can give this trail a *weight*, as described for the Keccak-p permutation in [4]. This will be explained in more detail in section 2.1.3.

2.1.2 Linear Cryptanalysis

In comparison to DC, *linear cryptanalysis* (LC) will compare the in- and outputs bit directly, instead of comparing their differentials [8]. However, we still try to find trails with high probability occurrences between the in- and output, where we can derive dependencies between the states [7]. Using these dependencies, we can again rate a trail by a weight, as explained in section 2.1.3.

2.1.3 Weight of a Trail

In order to rate the trails found, we give each differential trail a weight w , which were previously defined by the Keccak team [4]. This weight corresponds directly to the differential propagation probability of the trail, as $DP = 2^{-w}$, meaning that the weight increases with decreasing DP [5].

The weight of the whole trail Q is the sum of the weights of all differentials of the trail, which are called *restriction weights* w_r . These restriction weights are based on the differentials (b_{i-1}, a_i) , which are the Xoodoo states before (b_{i-1}) and after (a_i) the permutations of χ [8]. Based on these weights, we can estimate the differential propagation probability of that trail, giving the chance of the differentials $(\Delta_{in}, \Delta_{out})$ resulting in this specific trail. If

the weight of a trail is too low, the trail will have a high DP, making it easy to be exploited in attacks [5].

For this research are interested in 3-round trails Q , which are from the form:

$$Q = a_1 \xrightarrow{\lambda} b_1 \xrightarrow{\chi} a_2 \xrightarrow{\lambda} b_2 \xrightarrow{\chi} a_3, \quad (2.1)$$

Again, λ is defined by $\lambda\rho_{west} \circ \theta \circ \rho_{east}$, which are linear functions which use a shifting offset. The weight of these 3-round trails, which is specified by $w(Q)$, can be calculated by:

$$w(Q) = w(a_1) + w(b_1) + w(a_2) + w(b_2) + w(a_3) + w(b_3). \quad (2.2)$$

For our research purposes, we are searching for the low-weight trails for each set of shifting offsets. These trails reveal the most information about the underlying permutation and thus have to be avoided.

For the current 4 x 3 x 32 Xoodoo configuration the lower bound of the trail weight founds was 36 for 3-round trails [8]. With other words, the worst trail in the current Xoodoo implementation has weight 36.

2.2 Xoodoo Trails

For the Xoodoo trail analysis we used the Xoodoo software to generate trails, which was developed by the Keccak team [8]. The given software however was only able to generate n-round Xoodoo trails up to a given weight and print them afterwards, thus it had to be modified for this analysis (as described in 3.1).

For the generation of a 3-round trail, the program first generated a 2-round trail Q , which was afterwards extended to 3 rounds. These 3-round extended trails Q' can either be created by forward- or backward-extension [5]. For the forward extension, we first generate all trails compatible with $b_2 = \lambda(a_2)$. In the next step we use the resulting b_2 in order to append $a_3 = \chi(b_2)$. The resulting trail is a forward extended 3-round trail Q' , which has the earlier received 2-round trail up to b_2 as leading part.

In comparison to that, we can also extend a 2-round trail using backwards extension to a 3-round trail. For this, we first again create a 2-round trail denoted by $Q = (a_2, b_2, a_3)$, using the notation shown in 2.1. Afterwards we use χ^{-1} on a_2 in order to find $b_1 = \chi^{-1}(a_2)$. Then, we can calculate $\lambda^{-1}(b_1) = a_1$. With this, we can create the 3-round trail $Q' = (a_1, b_1, a_2, b_2, a_3)$, which has the previously taken Q as trailing part [5].

These different trails were created for many different Xoodoo configuration, iterating through all interesting shift offsets, which are shown in Table 3.1. A Xoodoo *configuration* is one specific set of shift offsets, used by θ , ρ_{east} , and ρ_{west} during the permutation round. These offsets specify, how

the bits in the Xoodoo are shifted during the rounds. A different set of shifting offsets could thus lead to a totally different set of trails. The original shifting offsets by the $4 \times 3 \times 32$ Xoodoo, which are used in Algorithm 1, are the vectors, specifying the bit shifts.

Additionally, for each set of offsets, the Xoodoo program created differential- and linear trails, including their corresponding weights.

All in all, leading to 4 different best lower bounds on the weight of the trails for each set of shifting variables, as it can be seen in Figure 3.2.

This produced an output file, with a listing of the lowest weight found for each Xoodoo configuration, including the method, with which the trail was found.

Between each step during the permutation rounds, the current output was analysed and compared to the previous as well as the next step. Depending on these outputs, each round was rated with a weight, which was determining how much these steps are depending on each other. In the following Figure 2.1, we can see a three round trail of a $4 \times 3 \times 8$ Xoodoo.

```

3-round differential trail core of total weight 36
Round 0 would have weight 8
Round 1 (weight 12):
NE      pE  S(K)    pW  NW
..... | ..... | .....
5..5.... | 5..5.... | 1..5..4.
..... | ..... | .....
5..5.... | 5..5.... | 1..5..4.
Round 2 (weight 16):
NE      pE  S(K)    pW  NW
..... | ..... | .....
5..4..5. | 5.....5. | 14.4..1.
..... | ..... | .....
5..1..5. | 5..5..5. | 14.5..5.

```

Figure 2.1: Screenshot of an example trail for a $4 \times 3 \times 8$ (96 bits) Xoodoo

When looking at this picture, we see three different states per round, where the first round is not shown. To bring this in perspective with Figure 1.1, we only see one plane from above here. However, the columns representing a 3-bit array, which can display the decimal numbers from 0 up to 7, such that 5 in the picture represents a column, where the lowest bit is 1, the bit in between is 0 and the top bit is also 1. So for the first part of round 1, there are only four columns out of the 32 which are not zero.

In Figure 2.1, we can already see one of the main problems with the current Xoodoo configuration. When the given inputs for the permutations rounds are chosen carefully to make this happen, some of the steps during the rounds does not have any impact at all.

The effect of θ depends on the active columns of the current state, as seen in Figure 1.2. If the column is active or not depends on the parity of that column. The *column-parity* is defined by the number of set bits in that specific column, where a column at the coordinates (x, z) is defined by the three bits (a_0, a_1, a_2) . A column is called active, if its parity is odd or 1.

This is the case, if there is exactly one active bit in the 3-bit array of that column. On the other hand, if the parity of a column is even or 0, it is inactive, which occurs when 0 or 2 bits of the 3-bit array are set.

In the example shown in Figure 2.1 we have four even columns in the first state of round 1, as all 4 columns, which have an entry, are represented by the 3-bit array (1,0,1). The worst case for the θ would occur, if all columns have an even parity (0), in which θ would be simply the identity-function, meaning it would not induce any new columns in the next step. When this event occurs, we say that the state is in the *column-parity kernel* [8]. In the optimal state each functions would induce as many bit columns as possible, keeping the diffusion of the state high and thus decreasing the DP of a specific trail.

For odd columns, starting from the column at (x, z) , the θ -function will induce this parity to two additional columns, namely $(x + 1, x + 5)$ and $(x + 1, z + 14)$. If it so happens, that these columns are already active in the current state, θ would also have less or even no impact, just by chance. In addition to that, if we add another active column at coordinates $(x, z + 9)$, it will have two impacts on the state. Firstly, it will induce an affected column at $(x + 1, z + 23)$, which is not too surprising. Secondly however, it will, together with the first active column coming from (x, y) cancel out the induced column at coordinates $(x + 1, z + 14)$. This effect is called a *run* [8]. If now, all columns in a state are odd, this run-effect will cancel out the whole θ -effect, inducing no additional new column. We call that event a *loop* [8]. This special cases cannot be avoided, as they are a consequence of the linear design of the function.

If we look closely to the first round of this three round trail, we can see that we basically have two times the same pattern. So this shows, that this specific trail is not a 96 bit trail but only two 48 bit trails, next to each other. An attacker could use knowledge for his advantage, to craft a larger width (with a factor of 2^n) Xoodoo version with the information gained from smaller ones, which would be less hard to find as it requires fewer calculations i.e. computing time. This is called the *Matryoshka effect* [4], which is a consequence of the design of the functions θ , ρ_{east} and ρ_{west} . All of these functions are shift-invariant, meaning that each bit-shift during the permutation round will impact the same bits relatively speaking for the size, no matter the width or length the current Xoodoo setup is based on, as long as it is dividable by a factor of 2.

Figure 2.2 illustrates how the toy-sized $4 \times 3 \times 8$ Xoodoo should look like on the left. Optimally it is a 3-dimensional matrix spanned by a (0/8) and a (4/0) vector. However, as the shifts constants are chosen in such a way, the right state of Figure 2.2 occurs. Instead of having a large $4 \times 3 \times 8$ Xoodoo, we only have two $2 \times 3 \times 8$ Xoodoo states next to each other, both spanned by a (0/8) and a (2/0) vector. This halves the weight of this specific trail if an adversary manages to identify this property. These symmetric properties

will be further explained in the next section.

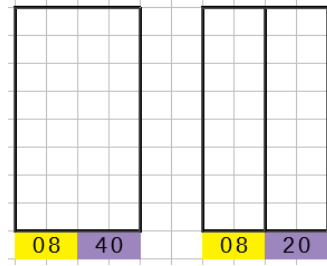


Figure 2.2: Symmetry in comparison, Left: Normal 8 x 4 Xoodoo - Right: Two symmetric 8 x 2 Xoodoo

2.3 Symmetry Properties

By design, the Xoodoo permutation offers a handful of symmetric schemes, which can be exploited by an attacker to eventually find trails for Xoodoo.

All of these symmetric state are not avoidable as 3 of the 5 steps (namely θ , ρ_{west} and ρ_{east}) during the round function operate in a symmetric way. These symmetric operations lead to the following schemes for a 96 bit Xoodoo, as seen in Figure 2.3, which are a consequence of the almost purely symmetric round function.

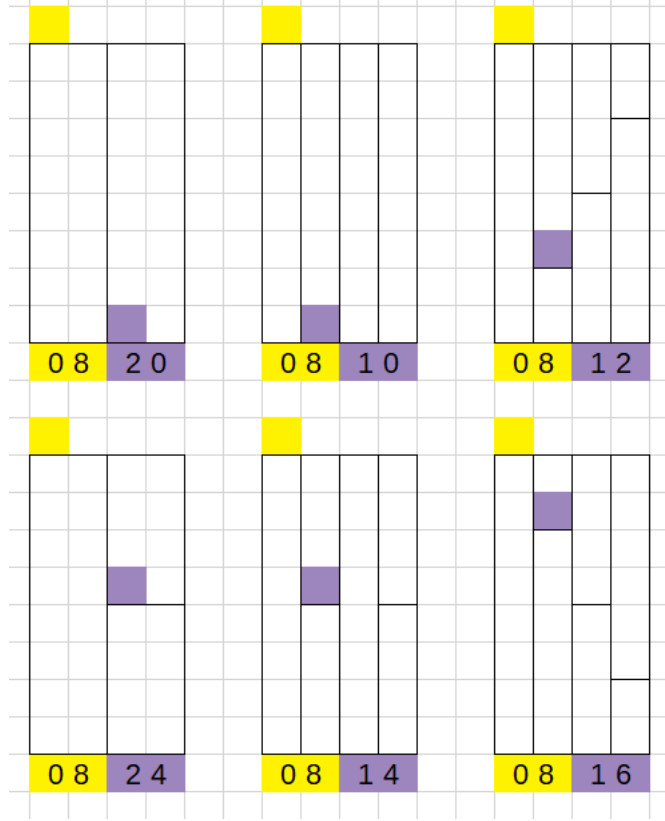


Figure 2.3: All possible symmetric setups of a 96 bit Xoodoo state

The above mentioned steps θ , ρ_{east} and ρ_{west} are all mostly using bit shifts for its permutations. One plane in Xoodoo can be seen as a infinite state, which is periodic in the x- as well as in the z-direction, with period 4 and period 32 respectively [8]. This leads to a loop, if the bit shifts of the permutation-steps reach the maximum width or length of the corresponding Xoodoo plane. These steps are invariant for translations on the 2-dimensional plane, if the vectors, which are used for the permutations, have the base vectors $(4, 0)$ and $(0, 32)$, which we call the *Xoodoo lattice* Ξ [8]. This Ξ defines the period of the infinite Xoodoo state, mentioned above. So the shifts along any vector in the Xoodoo lattice will map a Xoodoo state onto itself.

This invariance of the mentioned linear steps in the 2-dimensional (x, z) direction of Xoodoo results in a 2-dimensional Matryoshka property, which was explained earlier. In Xoodoo, this means that for a given state A and a lattice $v \in V$ (with $v = (x_v, z_v)$), we can express each column (x, y, z) in A with:

$$A[(x, y, z)] = A[(x, y, z) + v] = A[(x + x_v, y, z + z_v)], \quad (2.3)$$

which has two main consequences. This property holds for every column in the Xoodoo state A . If the given V is the original Xoodoo lattice Ξ , we are located in a normal Xoodoo state, with the same properties (width and length) as the initially given A . However, if the given V is a super-lattice of the Xoodoo lattice Ξ , the resulting state of 2.3 has additional symmetry in comparison to its original version [8].

As the $4 \times 3 \times 32$ Xoodoo state uses the base vectors $(4, 0)$ and $(0, 32)$, which are both a factor of 2, there are a high number of super-lattices of Ξ . These will each result in its own symmetrical smaller version of Xoodoo. This could be already seen in Figure 2.3 above, where the tested Xoodoo version of size $4 \times 3 \times 8$ already spanned six additional symmetrical versions, which even still have the same size of 96 bits. This is because each super-lattice V defines an additional symmetry class S_V , which is invariant to any translation in this lattice V [8], leading to fewer bit diffusions by the permutations steps. By the design of Xoodoo, we cannot find a second, different V' with $V \subset V'$ with corresponding symmetry class $S_{V'}$, such that a state invariant along V is also invariant along V' [8].

If we would define the symmetry classes by their lattices, we would end up with a total of 35 smaller versions with less bits, which are all symmetrical to the Xoodoo with 384 bits. An overview of all existing symmetry classes can be seen in table 2.1.

Number of symmetry classes	1	3	7	7	7	7	3	1
Number of Bits	384	192	96	48	24	12	6	3

Table 2.1: Number of Xoodoo States for different sizes

These symmetry properties can be used for the cryptanalysis done on the Xoodoo permutation. The main advantage of this is, that we do not have to scan as many trails, while still finding the weaker ones. This is a consequence of the fact, that all trails with a low weight, will have symmetric states in one or more rounds.

If we perform a cryptanalysis on these smaller versions of Xoodoo, we will not necessarily find the same trails as in the Xoodoo with 384. However, if we find a problematic low-weight-trail in the smaller version, we can conclude that a similar trail will also occur in the original $3 \times 4 \times 32$ Xoodoo version. This is consequence of the above explained Matroshka effect.

Chapter 3

Research

3.1 Adjustments on Xoodoo

In order to fulfil the research question, I added another function to the Xoodoo program (given in the Appendix A), which scans the given Xoodoo configuration until it finds a trail. For each configuration, the scan started with creating trails up to a weight of 16, which then was increased, until a trail with the given lowest weight was found. This weight then was noted in addition to the shift offsets, which were used to create this trail. The starting weight was set to 16, as we needed the weight to be as high as possible. If we would find a low weight trails, we would not use this setup anyway. After a trail was found, the next offset was tested, starting again with a weight of 16.

As we are looking for the best possible setup for our Xoodoo algorithm, the current scan is immediately stopped, when a trail is found, which is then saved to an output file. Using these, we can exclude configurations leading to low weight trails, such that we only keep the good shift offsets.

For this trail analysis we will use the symmetry properties of Xoodoo for our advantage. Using the super-lattices of the Xoodoo-lattice Ξ , we can construct a smaller symmetric state, with the size $1 \times 3 \times 32$, to limit the scan space. For this smaller Xoodoo version, it is much simpler to generate all possible trails and thus much faster to find a secure version with only good trails. This version of Xoodoo, which has only 1 lane (as x is 1), uses 1-dimensional shifting constant instead of the 2-dimensional shifting constants used in the $4 \times 3 \times 32$ Xoodoo.

3.2 Trail Analysis Results

In my tests, I iterated through around 33% of all possible interesting shifting offsets, used in the functions θ , ρ_{east} , and ρ_{west} and analysed the trails depending on their weight.

If a trail for the current configuration was found, the weight was noted and the next shifting offsets were tested. With these method, I listed all weights found for each configuration. For the different shifting offsets, I tried the values, as seen in Figure 3.1.

VariableName	UsedInFunction	Values
e_1	ρ_{east}	2 – 31
t_1	θ	4, 12, 20, 28
t_2	θ	3, 5, 7, \dots , 29
w_0	ρ_{west}	1 – 31
w_1	ρ_{west}	1 – 31

Figure 3.1: Different values for shifting offsets

This sums up to a total of around 1.6 million different possibilities. However, for each possible set of shifting offsets, the program created linear- as well as differential trails. Additionally, each setup has been tested using forward- and backward extension (see 2.2), giving each configuration a total of four different sets we have to test, increasing the number of total possibilities even further.

For each set of rotation offsets, the following output was generated:

```
32 Direct DC e1=4, t1=20, t2=13, w0=9, w1=23
36 Reverse DC e1=4, t1=20, t2=13, w0=9, w1=23
28 Direct LC e1=4, t1=20, t2=13, w0=9, w1=23
30 Reverse LC e1=4, t1=20, t2=13, w0=9, w1=23
Worst trail found has weight: 28
```

Figure 3.2: Example Output

For these specific shifting variables shown in the picture above, we have four different worst trail weights, namely 32, 36, 28 and 30. This means, that for this configuration, the worst trail has weight 28, even though the backwards-extended differential crypto analysis led to a worst trail weight of 36, which would be insanely good.

My tests were running for about three months and were stopped when $e_1 = 10$ was about half done. This means, that I only analysed about a third (34.4%) of the total space. For the tested space, I received the following results:

WorstTrailWeight	TimesFound	WorstTrailWeight	TimesFound
14	49.789	24	133.480
16	1.745	26	35.190
18	8.398	28	23.962
20	39.175	30	6.760
22	73.951	32	238

Figure 3.3: Number of different trails found

In addition to the results above, we found 1.312 trails of weight 34 and even 12 times a trail weight of 36. This were however for only one single trails and not for the whole setup (as shown in Figure 3.1). So we have 238 setup that results in a worst trail weight of 32.

The best setup, which was found during my tests is the one shown below.

```

34      Direct  DC      e1=4, t1=12, t2=17, w0=25, w1=4
34      Reverse DC      e1=4, t1=12, t2=17, w0=25, w1=4
32      Direct  LC      e1=4, t1=12, t2=17, w0=25, w1=4
34      Reverse LC      e1=4, t1=12, t2=17, w0=25, w1=4
Worst trail found has weight: 32

```

Using this setup, I found for three of the four cryptanalysis methods a lower trail bound of weight 34, only one of weight 32. This still results in a general trail bound of weight 32 for this specific Xoodoo configuration.

Using the symmetry present in Xoodoo, we can also conclude, that such a setup have to exist for the $3 \times 4 \times 32$ Xoodoo state. If we would extend this $1 \times 3 \times 32$ Xoodoo with 96 bits back to a full Xoodoo operating on 384 bits, we would multiply the weight we found by 4. This results in a weight of **128** for a 3 round Xoodoo permutation. Compared to the current implementation of Xoodoo, where the worst trail for 3 rounds has weight 36, the new worst trail found is 350% more efficient.

However, the configuration found cannot be directly used for the $3 \times 4 \times 32$ Xoodoo but rather shows, that there exists much more efficient setups than the current one. These will have to use different shifting variables than shown above, as 1-dimensional vectors cannot be used for the original Xoodoo.

Chapter 4

Conclusions

In this research, a crypto trail analysis on the 48-byte Xoodoo algorithm has been done in order to analyse its weaknesses and possibly prove the existence of an improved version. To do so, a smaller symmetric version of Xoodoo with only 96 bits has been analysed, which is much faster while still disclosing the weakest trails. Following the symmetry behind Xoodoo, we can use the knowledge gained about the lower-trail-bounds in order to show, that there exists a better $4 \times 3 \times 32$ Xoodoo configuration, which results in an increased lower bound for trails weights. This can be used to either increase the security level of the Xoodoo permutation, without changing the efficiency of the algorithm, when keeping the same amount of permutation rounds for the different constructions Xoodoo is used in. Additionally, as Xoodoo in its current state operates with six permutation rounds for e.g. the Xooff-deck-function, the number of rounds can be decreased in order to make the whole function much more efficient. The analysis has shown the of a much better Xoodoo versions, which could be achieved by using different shifting offsets during the permutation rounds.

Bibliography

- [1] Joan Daemen based on joint work with Bertoni et al. Innovations in permutation-based crypto, 2017. Presentation 21st Workshop on Elliptic Curve Cryptography Nijmegen <https://ecc2017.cs.ru.nl>, <https://ecc2017.cs.ru.nl/slides/ecc2017-daemen.pdf>.
- [2] G. Bertoni, J. Daemen, S. Hoffert, M. Peeters, G. Van Assche, and R. Van Keer. Farfalle: parallel permutation-based cryptography, 2017. <https://tosc.iacr.org/index.php/ToSC/article/view/855>.
- [3] G. Bertoni, J. Daemen, S. Hoffert, M. Peeters, G. Van Assche, and R. Van Keer. Xoodyak, a lightweight cryptographic scheme, 2019. <https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/round-1/spec-doc/Xoodyak-spec.pdf>.
- [4] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. The keccak reference, 2011. <https://keccak.team/files/Keccak-reference-3.0.pdf>.
- [5] J. Daemen and G. Van Assche. Differential propagation analysis of keccak, 2012. https://link.springer.com/chapter/10.1007/978-3-642-34047-5_24, In: Canteaut A. (eds) Fast Software Encryption. FSE 2012. Lecture Notes in Computer Science, vol 7549. Springer, Berlin, Heidelberg.
- [6] G. Bertoni et al. The making of keccak, 2014. <https://keccak.team/files/MakingOfKeccak.pdf>.
- [7] Howard M. Heys. A tutorial on linear and differential cryptanalysis. <https://pdfs.semanticscholar.org/8c58/e6cee8b31b8cd040d745b04e7f2f317122fd.pdf>.
- [8] Seth Hoffert Joan Daemen. The design of xoodoo and xoofff, 2018. Design of Xoodoo Permutation <https://eprint.iacr.org/2018/767.pdf>.

- [9] Seth Hoffert Joan Daemen. Xoodoo cookbook, 2019. Presentation of Xoodoo, <https://eprint.iacr.org/2018/767.pdf>.
- [10] Joan Daemen Silvia Mella. New techniques for trail bounds and application to differential trails in keccak, 2017. Differential Cryptanalysis, TrailWeight Analysis <https://eprint.iacr.org/2017/181.pdf>.

Appendix A

Appendix

A.1 Functions added

```
int initiateTrailSearch(int e1, int t1, int t2, int w1, int w0,
    int propagationType, bool backwardExtension, int
    extendWeight3R, unsigned int& minimumOffFour)
{
    const int directReverseImbalance = 2;
    const bool inKernel = true;
    const bool outOfKernel = true;
    const bool bareOnly = false;
    const bool doExtension = true;

    string tempName = "tempFile";

    int weightedWeight2R = (backwardExtension == false) ?
        extendWeight3R+directReverseImbalance :
        extendWeight3R-2-directReverseImbalance;
    XoodooParameters parameters(1, e1, t1, t2, w1, w0);

    unsigned int minWeight = 9999;
    ColoredBitSymmetryClass xoodoo(1, 32, parameters);

    cout << "*** " << xoodoo << endl;
    XoodooPropagation DCorLC(xoodoo, (XoodooPropagation::
        DCorLC)propagationType);
    ColoredBitSet bitSet(xoodoo, inKernel, outOfKernel,
        bareOnly);
    CoreGenerationCache cache(DCorLC);
    // Note: this is where the coefficients on the cost are
    set:
    CoreGenerationCostFunction cost(backwardExtension ? 1 :
        2, backwardExtension ? 2 : 1);
    TwoDimensionalHistogram hist(200, 200);
    unsigned int count = 0;
```

```

GenericTreeIterator<ColoredBit , ColoredBitSet ,
    XoodooPropagation , CoreGenerationCache ,
    TwoRoundTrailCoreFromColoredBits ,
    CoreGenerationCostFunction , GenericProgressDisplay>
// Note: this is where the limit on the generation of |X
| is set:
tree(bitSet , DCorLC, cost , weightedWeight2R);
{
    ofstream fout(tempName);

    while(!tree.isEnd()) {
        const TwoRoundTrailCoreFromColoredBits&
            core = (*tree);
        if (doExtension)
            extendTrailAll(fout , core ,
                backwardExtension , extendWeight3R ,
                minWeight);
        hist.hit(core.weights[0] , core.weights
            [1]);
        count++;
        ++tree;
    }
}

if(minWeight < minimumOfFour){
    minimumOfFour = minWeight;
}

return minWeight;
}

void printStats(int e1, int t1, int t2, int w1, int w0, int
    stats[2][2], ostream& rout, unsigned int minimumOfFour)
{
    for(int propagationType = 0; propagationType <= 1 ;
        propagationType++)
    {
        for(int direction = 0; direction <= 1; direction
            ++)
        {
            string result = std::to_string(stats[
                propagationType][direction])
            + ((direction == 0)? "\tDirect\t" : "\tReverse\t")
            + (propagationType ? "LC\t" : "DC\t")
            + "e1=" + std::to_string(e1)
            + ", t1=" + std::to_string(t1)
            + ", t2=" + std::to_string(t2)
            + ", w0=" + std::to_string(w0)
            + ", w1=" + std::to_string(w1) + "\n";

            rout << result;
        }
    }
}

```



```

        rout.flush();
    }
}
string allMin = "Worst trail found has weight: " + std::
    to_string(minimumOfFour) + "\n\n";
rout << allMin;
rout.flush();
}

```

```

void analyzeParameters(const CommandAndParameters& cap)
{
    string fileName = "maxWeight.txt";
    std::ofstream rout(fileName);
    try
    {
        const int extendWeight3R = cap.params[0];
        (void)cap;

        for(int e1 = 3; e1<=31; e1++) //e1 2-31
        {
            for(int t1 = 4; t1<=28; t1=t1+8) // t1 4, 12,
                18, 20, 28
            {
                for(int t2 = 3; t2 <= 29; t2 = t2 + 2) // t2
                    3,5,7,...,29
                {
                    for(int w1 = 1; w1 <=31; w1++) // w1 1,...,31
                    {
                        for(int w0 = 1; w0<=31; w0++) // w0 1,...,31
                        {
                            //Loop over propType and directions
                            unsigned int minimumOfFour = 9999;
                            int trailStats[2][2];
                            for(int propagationType = 0; (propagationType <=
                                1) ; propagationType++)
                            {
                                for(int direction = 0; (direction <= 1) ;
                                    direction++)
                                {
                                    {
                                        int runningWeight = extendWeight3R;
                                        bool backwardExtension = (direction == 1);
                                        int result = 9999;
                                        while(result == 9999)
                                        {
                                            result = initiateTrailSearch(e1, t1, t2, w1, w0,
                                                propagationType, backwardExtension,
                                                    runningWeight,
                                                        minimumOfFour
                                                    );
                                            trailStats[propagationType][direction] = result;
                                            if(result == 9999)
                                            {
                                                runningWeight += 2;
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}

```

```

        cout << "No trails found. Trying again with
                weight " + std::to_string(runningWeight) + "\n";
    }

    }
    }
    }
    //Print stats for current xoodoo configuration
    to output file
    printStats(e1, t1, t2, w1, w0, trailStats, rout,
               minimumOfFour);
    }
    }
    }
    }
}
catch(Exception e)
{
    cerr << e.reason << endl;
}
}

```