Radboud University Nijmegen

Faculty of Science

Analysis of Clustering Trails in Differentials over Iterative Permutations

Thesis MSc Cyber Security

Supervisor:
Joan Daemen

Author:
Carl Dworzack

Second reader:
Bert Mennink

December 2022

Abstract

With cryptographic permutations being relevant in many areas of cryptography, they have to be well-designed. An important aspect is the propagation of differences. We were interested in the following: given an input difference that leads to an output difference with relatively high probability following a certain path through an iterative, round-based permutation, we try to find other paths through the permutation with the same input and output differences. For the permutation Xoodoo this has already been done in a 3-round case. We look at 4-round permutations and also try to give an approach that is not specific to any permutation. To show the validity of our approach we implemented it and applied it to a 4-round Xoodoo case. Our program managed to find one additional path and also proves that there are no further paths. The run time was around one hour and 40 minutes so it was fairly efficient for this specific path.

Acknowledgment

This endeavour would not have been possible without the support from my supervisor Prof. Joan Daemen. He helped me to get a foothold in the subject matter, explained and resolved questions I had along the way and gave me feedback on how to improve all parts of this thesis.

I am also grateful to my friends Tara, Tsveti and Nour. They all looked over the thesis - some even multiple times - and their detailed feedback helped me hone this thesis in the final stages.

Lastly, I would like to mention my family and friends who aided me emotionally in the form of support, encouragement and regular inquiries into my progress thus motivating me further.

Contents

List of Figures

1 Introduction

A recent trend in the design of cryptographic schemes has been the use of permutation. These bijective functions are used, for example, within cryptographic encryption methods or hashing. Permutations are commonly built by having a relatively simple round function that is repeated multiple times. The round function itself consists of a few smaller linear and non-linear steps. The concepts described below have been introduced by Biham et al. [1].

An important observation to make about permutations is their difference distributions given a difference at the input. In other words, how often do certain differences at the output occur given two inputs with a fixed, initial difference, which is called the input difference. The output difference distribution is defined as how frequently an output difference occurs if all input pairs with in input difference are tried. Taking the input difference and output difference together results in a so-called differential. Determining the probability of a differential is difficult for well-designed, cryptographic permutations. For cryptographic purposes those with a high probability are of interest.

For a permutation a differential only states the differences at the beginning and the end. In an iterative, round-based permutation, there are, however, also intermediate differences after each of the iterative rounds. When the starting one, the intermediate ones and the final differences are all taken together, this is called a differential trail. To calculate the differential probability (DP) all pairs which have the same beginning, intermediate and end differences are counted. This number is then divided by 2^b, aka the number of all pairs. For a differential (which is only the first and last difference compared to a trail) the DP is calculated as the sum of the DPs of all trails that have the input and output difference of this differential. If multiple trails have the same first and last differences, we say that these trails cluster. In research, commonly only the DP of a trail is found; not, however, the DP of a differential which is more interesting. Higher DPs can be used for attacks, see [1]. In order to find this, we investigate clustering.

This is done by taking a trail and its corresponding differential and then looking for other trails that are different from the initial one, but still have the same beginning and end differences. If those are found, these trails then cluster. First, a general approach is taken before we look at the permutation Xoodoo designed by Daemen et al. [3] and its 4-round version in particular. Research [5] has already discovered multiple 4-round trails over Xoodoo and we will take a closer look at one of them. We develop a method to find clustering trails for a given 4-round differential which applies to Xoodoo and then use it on one trail. However, the techniques presented here can be used to search for clustering in any 4-round differential as long as their round function can be split into a linear and non-linear layer. Otherwise, this can serve as a starting point to analyse clustering of trails over more than 4 rounds. The search for clusters is based on an existing method for 3-round Xoodoo [2] where trails are found in a similar fashion. This allows us to formulate our research question.

Research Question: Is it possible to detect the existence of clustering trails efficiently for 4-round permutations given an initial trail?

1.1 Structure of this Thesis

Chapter 2 will first present an explanation of the permutation Xoodoo. Chapter 3 gives useful definitions, background and notations, specifically of differential cryptanalysis concepts. In Chapter 4 the search of the clustering trail cores will be explained. Chapter 5 then treats the trail core we used and the results we found. A conclusion is formed in Chapter 6.

2 Xoodoo

The following explanation of the Xoodoo permutation was taken from [3] and kindly provided by their authors:

Xoodoo is a family of permutations parameterized by its number of rounds n_r and denoted Xoodoo$[n_r]$.

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 3 lanes on top of each other and they are indexed by x. The Xoodoo state is illustrated in Figure 1.

The permutation consists of the iteration of a round function 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 2, the θ operation in Figure 3, the ρ_{east} and ρ_{west} operations in Figure 4.

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 [3] for the specification of the round constants for any index.

Finally, it is often useful to refer to only the bits - and their respective column. These will be indexed as (c, i) with c indicating the column in range 0 to 127 and i in range 0 to 2. The conversion from the three-dimensional indexing (x, y, z) is $i = y, c = 32x + z$.

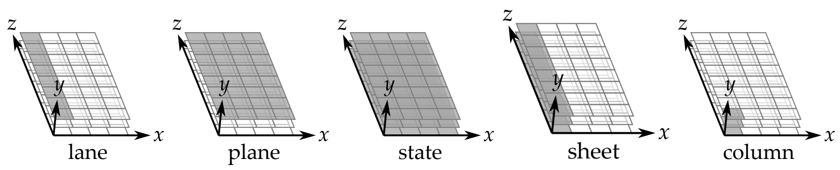


Figure 1: Toy version of the XOODOO state, with lanes reduced to 8 bits, and different parts of the state highlighted.

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 $A_{y'}$

Table 1: Notational conventions

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

```
**Parameters:** Number of rounds $n_r$ Round index $i$ from $1 - n_r$ to 0 $A = i(A)$
Here $i$ is specified by the following sequence of steps:
    $\theta$ :
        $P \leftarrow A_0 + A_1 + A_2$
        $E \leftarrow P \lll (1,5) + P \lll (1,14)$
        $A_y \leftarrow A_y + E$ for $y \in \{0,1,2\}$
    $\rho_{\text{west}}$ :
        $A_1 \leftarrow A_1 \lll (1,0)$
        $A_2 \leftarrow A_2 \lll (0,11)$
    $\iota$ :
        $A_0 \leftarrow A_0 + C_i$
    $\chi$ :
        $B_0 \leftarrow \overline{A_1} \cdot A_2$
        $B_1 \leftarrow \overline{A_2} \cdot A_0$
        $B_2 \leftarrow \overline{A_0} \cdot A_1$
        $A_y \leftarrow A_y + B_y$ for $y \in \{0,1,2\}$
    $\rho_{\text{east}}$ :
        $A_1 \leftarrow A_1 \lll (0,1)$
        $A_2 \leftarrow A_2 \lll (2,8)$
```

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

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

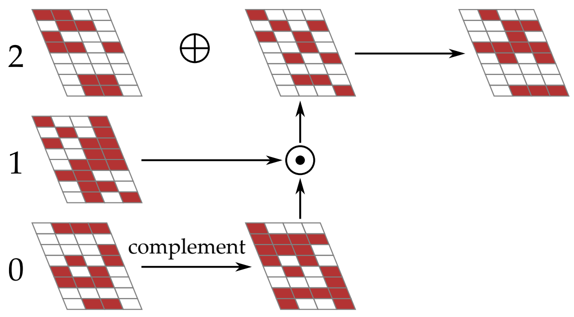


Figure 2: Effect of χ on one plane.

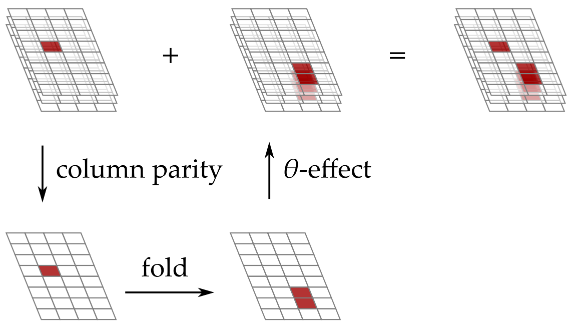


Figure 3: Effect of θ on a single-bit state.

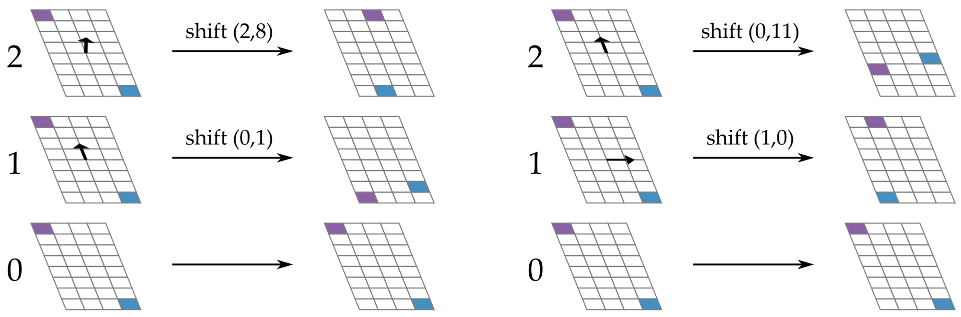


Figure 4: Illustration of ρ_{east} (left) and ρ_{west} (right).

3 Background and Basics

The following section will give some background information, definitions and concepts. First, some basics about linear algebra are introduced. Second, terminology for differential cryptanalysis is given. We introduce the concept of activity and then give some specific remarks about Xoodoo . Lastly, we give a short overview of a related paper.

3.1 Linear Algebra Basics

This section will give a brief introduction into linear algebra with regards to vector spaces and affine spaces. It is assumed that the reader is familiar with fields.

Definition 3.1. Let $\mathbb{F}$ be a field. Then, $(V, +, \cdot)$ with $+ : V \times V \longrightarrow V$ and $\cdot : \mathbb{F} \times V \longrightarrow V$ is a **vector space** over the field $\mathbb{F}$ if the following conditions are satisfied:

$$\begin{aligned}
&(V1)\ \ \forall u, v, w \in V \mid\ u + (v + w) = (u + v) + w,\\
&(V2)\ \ \forall u, v \in V \mid\ u + v = v + u,\\
&(V3)\ \ \exists e \in V \mid\ \forall v \in V \mid\ v + e = v,\\
&(V4)\ \ \forall v \in V \mid\ \exists - v \in V \mid\ v + (-v) = e,\\
&(V5)\ \ \forall a, b \in \mathbb{F}, v \in V \mid\ a \cdot (b \cdot v) = (a \cdot b) \cdot v,\\
&(V6)\ \ \exists 1 \in \mathbb{F}\ \mid\ \forall v \in V \mid\ 1 \cdot v = v,\\
&(V7)\ \ \forall a \in \mathbb{F}, u, v \in V \mid\ a \cdot (u + v) = a \cdot u + a \cdot v,\\
&(V8)\ \ \forall a, b \in \mathbb{F}, v \in V \mid\ (a + b) \cdot v = a \cdot v + b \cdot v
\end{aligned}$$

Note: While $\mathbb{F}$ can be any valid field, this thesis will exclusively deal with the case $\mathbb{F} = \mathbb{F}_2$ with the elements 0 and 1.

Elements $v \in V$ are referred to as **vectors**. In most cases these are a tuple consisting of multiple entries $v = (v_1, v_2, ...)$. According to the definition above, these vectors can be added to each other which is done per tuple entry using the field addition:

$$u + v = (u_1, u_2, ...) + (v_1, v_2, ...) = (u_1 + v_1, u_2 + v_2, ...) = (w_1, w_2, ...) = w, \ \ \forall u, v, w \in V$$

Similarly, they can also be multiplied by **scalars** (elements from the field). This is done by multiplying each tuple entry by the scalar using the field multiplication:

$$a \cdot v = a \cdot (v_1, v_2, ...) = (a \cdot v_1, a \cdot v_2, ...), \ \ \forall a \in \mathbb{F}, v \in V$$

Definition 3.2. The vector $e_i \in V$ of the form

$$e_i = (\underbrace{0, 0, ..., 0, 0}_{i-1}, 1, 0, 0, ..., 0, 0)$$

where only the i-th position is equal to 1 is called the i-th unit vector.

Definition 3.3. A **linear combination** of a set of vectors $F = (v_1, v_2, v_3, ...)$ with $v_i \in V$ is defined as

$$v = \sum_F c_i \cdot v_i$$

with the $c_i \in \mathbb{F}$. Since $\mathbb{F} = \mathbb{F}_2$ the coefficients are either 1 or 0 meaning that a vector is either used or not used for the construction of a new vector.

Definition 3.4. For a set of vectors $F = (v_1, v_2, v_3, ...)$ the **span** is defined as the set of all possible linear combinations:

$$span(F) = \left\{ v = \sum_F c_i \cdot v_i | c_i \in \mathbb{F} \right\}$$

Definition 3.5. A set of vectors $F = (v_1, v_2, v_3, ...)$ is called **minimal** if

$$|span(F)| > |span(F \backslash \{v\})| \;\; \forall v \in F.$$

Definition 3.6. A set of vectors $\mathcal{B} = (b_1, b_2, b_3, ...)$ is called **basis** of the vector space V if it is minimal and $span(\mathcal{B}) = V$

Definition 3.7. The **dimension** of a vector space is defined as $dim(V) = |\mathcal{B}|$. This is well-defined as all bases of a vector space have the same length.

From here on, only finite vector spaces will be looked at. Their bases can therefore be written as $\mathcal{B} = (b_1, ..., b_n)$ where $n = dim(V)$. A basis can be used to determine a vector space. Note that different bases can determine the same vector space.

Definition 3.8. Let $\mathbb{F}$ be a field. Then, $(A, +, \cdot)$ with $+ : A \times A \longrightarrow A$ and $\cdot : \mathbb{F} \times A \longrightarrow A$ is an **affine space** over the field $\mathbb{F}$ if the following conditions are satisfied:

$$\begin{aligned}
&(A1) \;\; \forall u, v, w \in A \;\mid\; u + (v + w) = (u + v) + w, \\
&(A2) \;\; \forall u, v \in A \;\mid\; u + v = v + u, \\
&(A3) \;\; \forall a, b \in \mathbb{F}, v \in A \;\mid\; a \cdot (b \cdot v) = (a \cdot b) \cdot v, \\
&(A4) \;\; \exists 1 \in \mathbb{F} \;\mid\; \forall v \in A \;\mid\; 1 \cdot v = v, \\
&(A5) \;\; \forall a \in \mathbb{F}, u, v \in A \;\mid\; a \cdot (u + v) = a \cdot u + a \cdot v, \\
&(A6) \;\; \forall a, b \in \mathbb{F}, v \in A \;\mid\; (a + b) \cdot v = a \cdot v + b \cdot v
\end{aligned}$$

This is a similar definition to that of the vector space with the exclusion of (V3) and (V4) meaning that every vector space is also an affine space. (V3) specifically means that the neutral element 0 was included in V, which is not necessarily the case for an affine space anymore. Therefore, an affine space cannot be described by only a basis since $0 \in span(\mathcal{B})$ for any $\mathcal{B} \subset V$.

Definition 3.9. An affine space A *can* be described by a basis $\mathcal{B} = (b_1, ...b_n)$ and an **offset** $\overline{a} \in A$ where

$$A = \overline{a} + span(\mathcal{B}) = \left\{ a \in V : a = \overline{a} + \sum_{\mathcal{B}} c_i \cdot b_i | c_i \in \mathbb{F} \right\}$$

Lemma 3.10. *Applying a linear function λ to all elements of a basis $\mathcal{B}$ and an offset $\overline{a}$ of an affine space results in a new basis and offset defining a new affine space.*

Proof. All six properties (A1)-(A6) are trivially fulfilled. Further, $\lambda(\mathcal{B})$ is still minimal. Assume it is not, so one of the new basis vectors $\lambda(b)$ would be a linear combination of the others. Then, its preimage would also be a linear combination of the initial basis $\mathcal{B} \backslash \{b\}$ since λ is a linear function. This would be a contradiction to $\mathcal{B}$ being a basis. Therefore, we have a new affine space with offset $\lambda(\overline{a})$ and basis $\lambda(\mathcal{B})$. $\square$

Note that the same is true for vector spaces.

3.2 Differential Cryptanalysis

The following concepts were introduced in and taken from [1]. One of the aspects investigated in differential cryptanalysis are cryptographic permutations $f : \mathbb{F}_2^b \longrightarrow \mathbb{F}_2^b$ to quantify the propagation of differences. For this, the permutation is used on an input pair with a fixed input difference. This results in an output difference. Together, the input and output differences $(\Delta_{\text{in}}, \Delta_{\text{out}}) \in (\mathbb{F}_2^b)^2$ are called a differential. The inputs $x \in \mathbb{F}_2^b$ to the permutation f that follow this differential are defined as

$$U_f(\Delta_{in}, \Delta_{out}) := \{x \in \mathbb{F}_2^b | f(x) + f(x + \Delta_{in}) = \Delta_{out}\}$$

which is called the **solution set**. Further, it is said that Δ_{in} is compatible with Δ_{out} through f if $Pr\,[f(x) + f(x + \Delta_{in}) = \Delta_{out}] > 0$. To determine the likelihood of a differential the notion of the **differential probability** (DP) is introduced as follows:

$$DP_f(\Delta_{in}, \Delta_{out}) := \frac{|U_f(\Delta_{in}, \Delta_{out})|}{2^b},$$

aka the number of elements in the solution set divided by the number of all possible b-bit string inputs. However, the more rounds a permutation has and the wider it is, aka the larger the parameter b, the less feasible it becomes to calculate the DP by finding the solution set. The computational effort would be in $\mathcal{O}(exp(b))$.

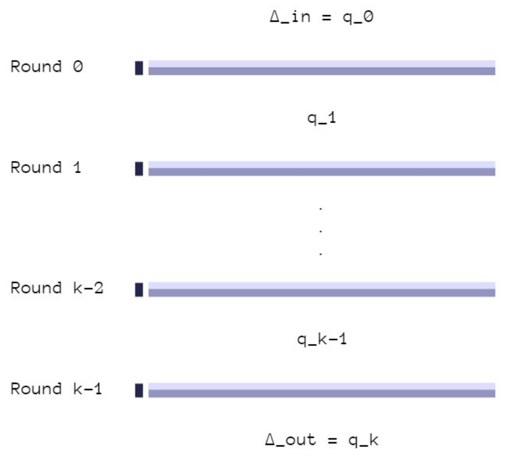


Figure 5: Shematic of intermediate differences over a k-round trail

For iterative, round based permutations it can be useful to not only look at the input and output difference but also the intermediate differences as shown in Figure 3.2. The sequence $Q = (q_0, q_1, ..., q_{k-1}, q_k) \in (\mathbb{F}_2^b)^{k+1}$ including all these differences is called a **k-round differential trail** if $DP_i(q_i, q_{i+1}) > 0 \ \ \forall i \in \{0, ..., k-1\}$. In other words, all these round differentials must have a non-zero chance of occurring. We say a trail Q **is in** a differential $(\Delta_{in}, \Delta_{out})$ if $q_o = \Delta_{in}$ and $q_k = \Delta_{out}$.

We can define the DP of a trail similar to the DP of a differential: It is the fraction of all input differences that follow the trail with each intermediate difference q_i divided by all possible b-bit inputs.

Definition 3.11. For a single trail $Q = (q_0, ..., q_k) \in (\mathbb{F}_2^b)^{k+1}$ its **expected differential probability** (EDP) is defined as

$$EDP_f(Q) = \prod_{i=0}^{k-1} DP_i(q_i, q_{i+1}).$$

The EDP is often a good approximation of the DP. The EDP is the same as the DP if all the round differentials are independent:

$$\begin{aligned} DP_i(q_i, q_{i+1}) &= Pr[\text{output difference} = q_{i+1} \mid \text{input difference} = q_i] \\ &= \frac{Pr[\text{input difference} = q_i \wedge \text{output difference} = q_{i+1}]}{Pr[\text{input difference} = q_i]} \\ &= \frac{Pr[\text{input difference} = q_i] \cdot Pr[\text{output difference} = q_{i+1}]}{Pr[\text{input difference} = q_i]} \\ &= Pr[\text{output difference} = q_{i+1}] \end{aligned}$$

For certain cases, this approximation can become exact if all the individual round differentials are independent. Xoodoo , the permutation looked at later, is one of those cases. We will introduce the following instead of using the DP:

The set of all differential trails that are in one differential $(\Delta_{in}, \Delta_{out})$ is defined as

$$DT_f(\Delta_{in}, \Delta_{out}) := \left\{Q \in (\mathbb{F}_2^b)^{k+1} | q_0 = \Delta_{in} \wedge q_k = \Delta_{out}\right\}$$

Definition 3.12. The EDP of a differential $(\Delta_{in}, \Delta_{out})$ is then defined as

$$EDP_f(\Delta_{in}, \Delta_{out}) = \sum_{Q \in DT_f(\Delta_{\text{in}}, \Delta_{\text{out}})} EDP_f(Q),$$

which is ultimately what we are interested in. Note that the DP of the differential can be computed by adding up the DPs of all trails in $DT_f(\Delta_{\text{in}}, \Delta_{\text{out}})$. In the case that $|DT_f(\Delta_{in}, \Delta_{out})| > 1$ those trails are said to **cluster**. Omitting the first and last differences $q_o = \Delta_{in}$ and $q_k = \Delta_{out}$ leaves the **differential trail core** $(q_1, ..., q_{k-1})$. A trail core therefore corresponds to a set of trails with the same inner differences but other input and output differences.

Definition 3.13. For a differential $(\Delta_{\text{in}}, \Delta_{\text{out}})$, its **restriction weight** $\text{w}(\Delta_{\text{in}}, \Delta_{\text{out}})$ is defined as

$$\text{w}(\Delta_{\text{in}}, \Delta_{\text{out}}) := -log_2(DP_f(\Delta_{\text{in}}, \Delta_{\text{out}})).$$

Further, for a k-round differential trail $Q = (q_0, ..., q_k)$ the restriction weight is defined as

$$\text{w}(Q) = \sum_{i=0}^{k-1} \text{w}(q_i, q_{i+1}).$$

The **weight profile** $\text{w}_p(Q)$ of a k-round trail Q is defined as the sequence

$$\text{w}_p(Q) := \left(\text{w}(b_i, a_{i+1})\right)_{i \in \{0,\dots,k-1\}}.$$

In cryptography a **Substitution-Box**, or **S-Box** for short, is a bijective mapping between bit strings of a certain length, the width m of the S-Box. Normally, these S-Boxes are non-linear and have no mapping that can be easily expressed mathematically. They are used for example in encryption algorithms like Rijndael [4] or in permutations like Xoodoo in this case.

One S-Box only maps m bits onto m new bits so it operates on the column level. By applying an S-Box to all columns in a vector the non-linear S-Box layer is created. In the following section this non-linear layer will be referred to a μ.

For every S-Box, the bit string of just zeros is mapped to itself. This means that inactive columns (ones with just zeros) stay inactive columns and active columns stay active columns. The activity pattern of a vector is therefore invariant under the S-Box layer!

Definition 3.14. Two columns c_1, c_2 are said to be **compatible** through the non-linear layer, if $DP(c_1, c_2) > 0$.

3.3 Activity and Activity Patterns

This section now goes into more detail on how to look for clustering trails in a k-round, iterative permutation. These permutations are commonly constructed of a linear and a non-linear layer in each round which are applied one after another. The linear part will be noted as λ and the non-linear part as μ.

With the separation in linear and non-linear layer, a trail can be described in more detail as $(a_0, b_0, a_1, b_1, ..., a_{k-1}, b_{k-1}, a_k)$ where $\lambda(a_i) = b_i$. The potential cluster trail will be noted as $(a_0^*, b_0^*, a_1^*, b_1^*, ..., a_{k-1}^*, b_{k-1}^*, a_k^*)$. Given is only a trail core meaning it would be $(a_1, b_1, ..., a_{k-1}, b_{k-1})$. We need the following definitions about activity and compatibility to proceed:

Definition 3.15. A bit position (c, i) in a vector v is called **active**, if $v(c, i) = 1$. It will be written as $act_v(c, i) = 1$ or $act_v(c, i) = 0$. The latter indicates the bit position being **inactive**.

Definition 3.16. A column position (c) in a vector v is called **active**, if one of the bit positions in this column is active in v. It will be written as $act_v(c) = 1$ or $act_v(c) = 0$. The latter indicates the column position being **inactive**.

Definition 3.17. The **column activity pattern** a of a vector v is another vector indicating which of the column positions is active in the first one. It will be referred to as **activity pattern** for the rest of this thesis.

Definition 3.18. The number of active columns in a vector v is written as $\#_{act}(v)$.

Building on these activity definitions for vectors we can now expand them to more generally talk about activity in affine spaces.

Definition 3.19. A bit position (c, i) in an affine space V is called **active**, if there are vectors in the affine space for which the bit at this position is active. It will be written as $act_V(c, i) = 1$ or $act_V(c, i) = 0$. The latter indicates the bit position being **inactive** in the affine space.

Definition 3.20. A column position c in an affine space V is called **active**, if there are vectors in the affine space for which any of the bit positions in this column are active. It will be written as $act_V(c) = 1$ or $act_V(c) = 0$. The latter indicates the column position being **inactive** in the affine space.

Definition 3.21. Given a vector v, its **activity matching set** $\mathcal{S}_{act}(v)$ is defined as all vectors that have the same activity pattern as v.

This activity matching set can be expanded to be a vector space that includes the activity matching set.

Definition 3.22. For a vector v and its activity matching set $\mathcal{S}_{act}(v)$, the **expanded activity matching space** $\mathcal{V}_{actM}(v)$ is defined as the vector space of lowest dimension that includes all elements of $\mathcal{S}_{act}(v)$

Since this vector space is a proper super set of $\mathcal{S}_{act}(v)$ it also includes elements that have a different activity pattern than v. How this vector space is constructed, what its basis looks like, and of what dimension it is, will all be explained in Section 4.1.

3.4 Specifics about Xoodoo

This section will mention and explain a few things that are specific to the permutation XOODOO that we are looking at here. If the ideas of this paper are applied to other permutations, the conclusions formed in this chapter cannot always be used.

The specific S-Box used in the XOODOO permutation has a width $m = 3$ meaning it operated on the 3-bit wide columns. Definition 3.14 can be made more concrete in the case of XOODOO and leads to the following observation: Two columns c_1, c_2 are compatible through the non-linear layer χ if they are both inactive or if their *logical or* has an uneven number of active bits. All active, compatible pairs are shown in Table 3.

c_1	(0, 0, 1)	(0, 1, 0)	(1, 0, 0)	(0, 1, 1)	(1, 0, 1)	(1, 1, 0)	(1, 1, 1)
c_2	(0, 0, 1)	(0, 1, 0)	(1, 0, 0)	(0, 0, 1)	(0, 0, 1)	(0, 1, 0)	(0, 0, 1)
	(1, 0, 1)	(1, 1, 0)	(1, 1, 0)	(0, 1, 0)	(1, 0, 0)	(1, 0, 0)	(0, 1, 0)
	(0, 1, 1)	(0, 1, 1)	(1, 0, 1)	(1, 0, 1)	(0, 1, 1)	(0, 1, 1)	(1, 0, 0)
	(1, 1, 1)	(1, 1, 1)	(1, 1, 1)	(1, 1, 0)	(1, 1, 0)	(1, 0, 1)	(1, 1, 1)

Table 3:
All compatible columns c_2 given an active column c_1 through the non-linear layer χ

We can see that there are always four matching columns and we want to express them in a more structured way than as a set of four elements. For every case, these four elements can be described by an affine space of dimension two. In other words, by one offset vector and a basis consisting of two basis vectors. This is shown individually for each column value in Table 4.

Column	Matching Affine Space
(0, 0, 1)	(0, 0, 1) + α (0, 1, 0) + β (1, 0, 0)
(0, 1, 0)	(0, 1, 0) + α (0, 0, 1) + β (1, 0, 0)
(1, 0, 0)	(1, 0, 0) + α (0, 0, 1) + β (0, 1, 0)
(0, 1, 1)	(0, 1, 0) + α (1, 0, 0) + β (0, 1, 1)
(1, 0, 1)	(0, 0, 1) + α (0, 1, 0) + β (1, 0, 1)
(1, 1, 0)	(1, 0, 0) + α (0, 0, 1) + β (1, 1, 0)
(1, 1, 1)	(1, 0, 0) + α (0, 1, 1) + β (1, 0, 1)

Table 4: Columns and compatible affine spaces through χ

The S-Box only operates on the column level, so this only described the potential columns. However, the same thoughts apply when moving to the S-Box layer, aka χ. If we have a vector v, then we can use this to create an affine space including all potential vectors that are compatible with v through χ.

Definition 3.23. Given a vector v, its **activity matching affine space** $\mathcal{A}_{act}(v)$ is defined as all vectors that are compatible with v through the non-linear layer χ.

Here we give an algorithm for the construction of $\mathcal{A}_{act}(v)$. We start with an empty basis and an offset that is completely set to zero. We iterate over every column c in v. For each column value $v(c)$, we get the affine space that describes the compatible columns through χ via the function *affineColumn* as shown in Table 4. This function is described in detail in Appendix A. As this only gives the column values, we pad them with zeros to complete vectors. The two basis vectors are added to the basis and the offset is added to the offset. This process is shown in Algorithm 3.4.

Algorithm 2 Pseudo Code for Creating the Activity Matching Affine Space

```
Require: $v$
  $\mathcal{V}$ is empty basis
  $\overline{v}$ is offset and set to be 0
  for active $v(c) \in v$ do
      $c_o, c_1, c_2 \leftarrow$affineColumn($v(c)$)        ▷ gives values from Table 4. See Appendix A
      $\overline{v} \leftarrow \overline{v} + (\underbrace{0, 0, ..., 0, 0}_{3c\ times}, c_o, 0, ..., 0)$
      add to $V$: $(\underbrace{0, 0, ..., 0, 0}_{3c\ times}, c_1, 0, ..., 0)$
      add to $V$: $(\underbrace{0, 0, ..., 0, 0}_{3c\ times}, c_2, 0, ..., 0)$
```

Since 2 basis vectors are added to the basis for every active column c in v, the dimension of the affine space V will end up as $dim(V) = 2 \cdot \#_{act}(v)$.

These explanations were under the assumption, that we have a given v that χ is applied to. But since the S-Box is symmetric, the χ is also symmetric. Therefore, we can just as well describe all possible vectors u that are compatible with a given vector v via the connection $\chi(u) = v$.

One last thing to note, is that for the XOODOO permutation, for a state v its restriction weight $\mathrm{w}(v)$ is equal to twice the number of active columns. Therefore, $\mathrm{w}(v) = 2 \cdot \#_{act}(v)$

3.5 Related and Previous Work

In their paper *Thinking Outside the Superbox* [2] Bordes et al. formalize and analyse the notion of alignment for different cryptographic primitives.

They also relate this concept to clustering of trail cores and specifically treat 3-round trail cores for XOODOO . Briefly summarized, their approach is as follows: For a given trail core (a_1, b_1, a_2, b_2), they create two vector spaces $U = \mathcal{V}_{actM}(a_1)$ and $V' = \mathcal{V}_{actM}(b_2)$. Applying the linear layer λ to the elements of $\mathcal{U}$ and its inverse λ^{-1} to $\mathcal{V}'$ gives the vector spaces U' and V.

To find a clustering trail core, they now need a pair $(u', v) \in U' \times V$ with $\chi(u') = v$. Instead of checking all potential pairs, they instead eliminate some basis elements in $\mathcal{U}'$ and $\mathcal{V}$. Those basis vectors could never contribute to a vector that belongs to a matching pair (u', v). Once the bases were reduced thusly, they checked the remaining pairs one by one.

We reuse the process of thinning out the bases in our approach as well. However, we adapted it to fit out situation and refined it to thin our the bases even further.

4 Analysis

This section describes the process of searching for clustering trails. We will use a given trail core $(a_1, b_1, a_2, b_2, a_3, b_3)$. Important here are only the input difference a_1 and the output difference b_3. We start by first describing the general idea and then going into more detail on all the individual steps.

4.1 Setup and First Steps

An overview of the 4-round version of XOODOO can be seen in Figure 6. Recall that a round in XOODOO consist of the steps $\theta, \rho_{west}, \iota, \chi, \rho_{east}$. For convenience, we will rephrase XOODOO to start with ρ_{east} leading to the new order $\rho_{east}, \theta, \rho_{west}, \iota, \chi$. That way we get a clear separation in the linear layer $\lambda = \iota \circ \rho_{west} \circ \theta \circ \rho_{east}$ and the non-linear layer χ.

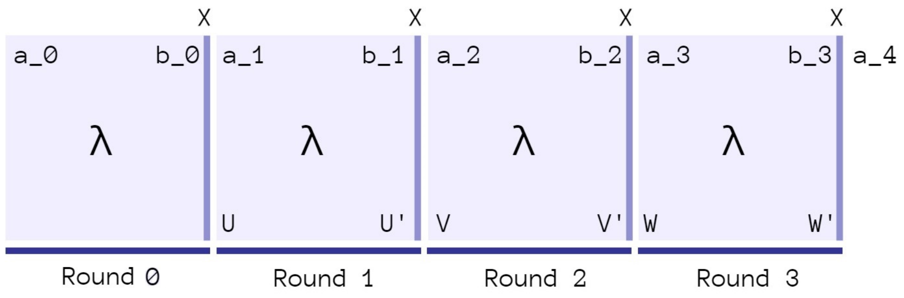


Figure 6: Overview of 4 Round Xoodoo Trail $a_0, b_0, a_1, b_1, a_2, b_2, a_3, b_3, a_4$:
λ indicates the linear layer, X the non-linear layer χ. U, U', V, V', W and W' are explained further below (Table 4.1)

The overall approach to finding clustering trail cores $(a_1^*, b_1^*, a_2^*, b_2^*, a_3^*, b_3^*)$ to the given trial core is as follows: We use $U := \mathcal{V}_{actM}(a_1)$ to include at least all potential values of a_1^*. Since U is a vector space, we can apply the linear layer λ to its basis elements $u \in \mathcal{U}$ which results in a new basis $\mathcal{U}'$ for the vector space U'.

Similarly, we can create $W' = \mathcal{S}_{act}(b_3)$, which gives us exactly all potential values for b_3^*. We iterate over all elements in $w' \in W'$ like this: Obtain $w = \lambda^{-1}(w')$. This now represents a potential a_3^*. We now create the activity matching affine space $V' := \mathcal{A}_{act}(w)$, which includes all potential b_2^*. This gives a vector space U' and can use linear transformation of $\mathcal{V}'$ to also get the affine space V.

We now want to find a pair $(u', v) \in U' \times V$ that is compatible through the non-linear layer. For this, we eliminate elements from the bases $\mathcal{U}'$ and $\mathcal{V}$ that could never lead to such pairs. All potential pairs that are left after this are then checked in the end. If we find a compatible one where $act(\lambda^{-1}(u')) = act(a_1)$, we have a clustering trail core $(\lambda^{-1}(u'), u', v, \lambda(v), w, w')$.

The last check for activity is needed because we used $U = \mathcal{V}_{actM}(a_1)$ in the beginning. This included some elements that actually do not have the right activity pattern. For $W' = \mathcal{S}_{act}(b_3)$, this is not needed as we have only matching activity patterns. A pseudo code description of this is given in Algorithm 3.

An overview of the sets U, U', V, V', W, W', their structures and how they are obtained can be found in Table 5. Their naming scheme is alphabetically with those after the linear layer having an added *prime* to their name.

For a potential trail to cluster with the given one, they must be part of the same differential per definition. Therefore, $a_0 = a_0^* = \Delta_{in}$ and $a_4 = a_4^* = \Delta_{out}$. This also

Algorithm 3 Pseudo Code for 4-Round Analysis:
Constructing $\mathcal{V}_{actM}(\cdot)$ is explained in Algorithm 4.1, constructing $\mathcal{A}_{act}(\cdot)$ is explained in Algorithm 3.4, *thinOut* is explained in Section 4.2.

```
Require: $a_1, b_3$
  $U \leftarrow \mathcal{V}_{actM}(a_1)$
  $\mathcal{U}' \leftarrow \lambda(\mathcal{U})$
  $W' \leftarrow \mathcal{S}_{act}(b_3)$
  for $w' \in W'$ do
      $w \leftarrow \lambda^{-1}(w')$
      $V' \leftarrow \mathcal{A}_{act}(w)$
      $\mathcal{V} \leftarrow \lambda(\mathcal{V}')$
      $(\mathcal{U}', \mathcal{V}) \leftarrow$thinOut$(\mathcal{U}', \mathcal{V})$
      for $u' \in U'$ and $v \in V$ do
          if $v$ and $u'$ are compatible then
              $u \leftarrow \lambda^{-1}(u')$
              if $act_u = act_{a_1}$ then
                  output $(u, u', v, \lambda(v), w, w')$
              end if
          end if
```

	Name	Obtaining	Type	Name	Obtaining	Type
Round 1	U	$\mathcal{V}_{actM}(a_1)$	vector	U'	$\lambda(\mathcal{U})$	vector
Round 2	V	$\lambda^{-1}(\mathcal{V}')$	affine	V'	$A_{S-Box}(w)$	affine
Round 3	W	$\lambda^{-1}(W')$	set	W	$\mathcal{S}_{act}(b_3)$	set

Table 5: Overview of sets, vector spaces and affine spaces

extends to $b_0 = \lambda(a_0) = \lambda(a_0^*) = b_0^*$. So we are just looking for a trail core.

Any trail core that could potentially cluster with the given one has to have an a_1^* that is compatible with b_0 through the non-linear layer. And since the activity pattern is invariant under the S-Box layer, we can say that all potential a_1^* must have the same activity pattern as b_0 and consequently as a_1. We can therefore express all potential a_1^* via the activity matching set $\mathcal{S}_{act}(a_1)$. However, since we later want a vector space structure, we are going to choose $U = \mathcal{V}_{actM}(a_1)$ here to show all potential a_1^*. This includes more elements than $\mathcal{S}_{act}(a_1)$, meaning we now have some $u \in U$ with $act(u) \neq act(a_1)$.

Algorithm 4 Pseudo Code for Basis Creation of Expanded Activity Matching Space

```
Require: $v$
  $\mathcal{V}$ is empty basis
  $a \leftarrow$activityPattern$(v)$
  for $a(c) \in a$ do
      if $a(c) = 1$ then
          add $e_{3\cdot c}, e_{3\cdot c+1}, e_{3\cdot c+2}$ to $\mathcal{V}$
      end if
```

The construction of $\mathcal{V}_{actM}(a_1)$ is shown in Algorithm 4.1 and works as follows: The activity pattern of v is evaluated. For each column c that is active, the vector space has to include the seven active variations for the 3-bit column. This can be achieved by using the three unit vectors e_{3c}, e_{3c+1} and e_{3c+2}. Via linear combinations, all possible column values can be created. The case $(0,0,0)$ is now also included. These are the extra elements alluded to before. Since three basis vectors are added for each active column, it follows that $dim(\mathcal{V}_{actM}(a_1) = 3 \cdot \#_{act}(a_1)$. Further, we can say that $|\mathcal{V}_{actM}(v)| = 2^{3 \cdot \#_{act}(a_1)} = 8^{\#_{act}(a_1)}$. More generally, for an S-Box that has width m, it is $|\mathcal{V}_{actM}(v) = 2^{m \cdot \#_{act}(a_1)}|$.

The *extra* elements in $\mathcal{V}_{actM}(a_1)$ can be quantified by looking at $|\mathcal{S}_{act}(a_1)$: Each column that is active can have one of 7 different values; the inactive $(0,0,0)$ case is excluded from the eight combinatorial possibilities. Since there are $\#_{act}(v)$ active columns in v, there have to be $7^{\#_{act}(v)}$ elements in total. And more general, for an S-Box that has width m, we have $|\mathcal{S}_{act}(v)| = (2^m - 1)^{\#_{act}(v)}$. The number of extra elements is therefore $8^{\#_{act}(a_1)} - 7^{\#_{act}(a_1)}$

We obtain the vector space U' that represents at least all potential b_1^* by using the basis $\mathcal{U}$ of U. Applying the linear layer to every basis element gives the new basis $\mathcal{U}'$ of the vector space U'.

Following the same logic as for a_1, every b_3^* must have the same activity pattern as a_4. Which, in turn, is the same as that of b_3. So we again represent all potential b_3^* via the set $W' = \mathcal{S}_{act}(b_3)$. Unlike for representing the differences for the first round in U and U', we do not need a vector space structure for the differences of Round 3 represented via W'.

We now have all potential differences in Round 1 as well as Round 3. The differences of Round 2 depend on what the differences are in Round 1 and 3. Since it would be too much to try and do all of them at the same time, we iterate over the elements in W' one by one. We then check whether there are any clustering trails given this $w' \in W'$, before checking the next one and so on.

Hence, a fixed $w' \in W'$ is picked and converted via $w = \lambda^{-1}(w')$. Next, we are interested in all the potential differences that would be compatible with this w through the non-linear layer. These potential differences form an affine space (see Section 3.4), which we will use as $V' := \mathcal{A}_{act}(w)$. Applying the inverse of the linear layer to its basis $\mathcal{V}'$ gives us the basis $\mathcal{V}$ of the affine space V. This affine space now includes exactly all potential differences a_2^* given that we picked $b_3^* = w'$.

Together with V we now have a vector space U' showing at least all potential values b_1^*. To find a clustering trail core, we need a compatible pair $(u', v) \in U' \times V$. Instead of checking all potential pairs individually, the following optimization was taken.

4.2 Thinning out U' and V

The goal of this step is to reduce the spaces U' and V by manipulating their bases to thereby reduce the number of potential pairs (u', v) that need to be checked for being compatible or not. This is done by repeating three steps, each aimed at removing basis vectors until no more can be taken away.

Step One - Direct Elimination: Every column c is checked for the case that

$$act_{U'}(c) = 0 \wedge act_{V}(c) = 0 \wedge act_{\overline{v}}(c) = 1$$

This means, every $v \in V$ would be active in column c. However, every $u' \in U'$ is inactive in that column. Therefore, every $v \in V$ has an activity pattern with an active column at this position while every $u' inU'$ has an activity pattern that is inactive in that column. So none of the pairs $(u', v) \in U' \times V$ can ever be compatible though χ. Hence, the picked $w' in'$ leads to no compatible pairs.

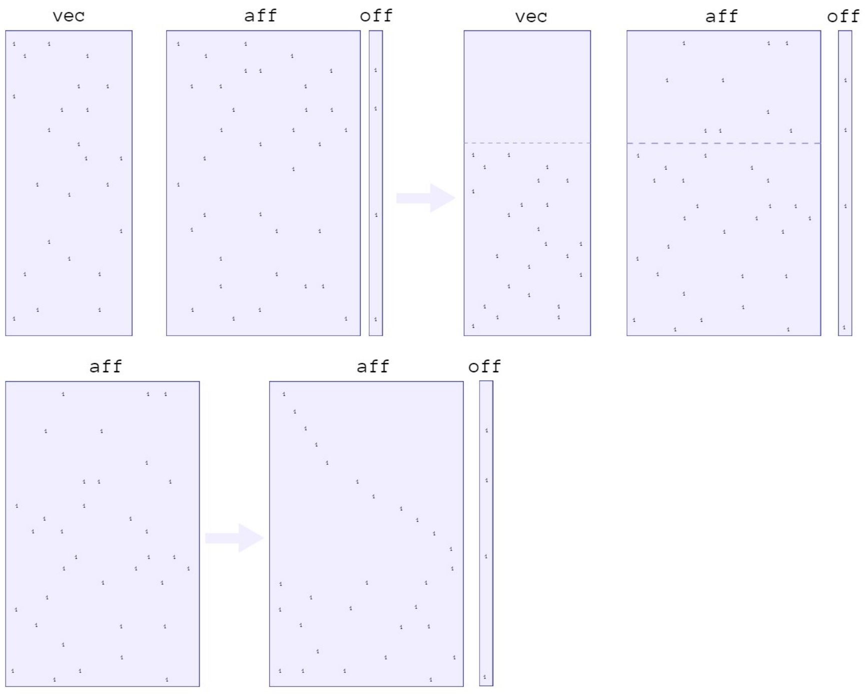


Figure 7: Overview of Step Two of Thinning Out
vec is the basis $\mathcal{U}'$ of vector space U', *aff* the basis $\mathcal{V}$ of the affine space V and *off* the corresponding offset $\overline{v}$. The top row shows the reordering of bit positions according to activity in U'. The bottom row shows the triangulization of $\mathcal{V}$.

Step Two - Isolated Bits: This step uses the same criteria for elimination as the thinning out process by Bordes et al. [2], which we mentioned in Section 3.5. The original version worked on two vector spaces, so we adapted it slightly to with a vector space and an affine space instead.

Definition 4.1. A bit position (c, i) of a vector b in a basis $\mathcal{B}$ is said to be an **isolated active bit** if $b(c, i) = 1$ and $\tilde{b}(c, i) = 0$ for all $\tilde{b} \in \mathcal{B} \backslash \{b\}$.

This definition is given for vector spaces. However, it can also be applied to affine spaces. The definition stays the same with potentially a small manipulation of the offset taking place. If the offset is active in that bit position, we create a new offset by adding the basis vector b to the old offset. This means we have an offset that is guaranteed to be inactive in the relevant bit position (c, i).

We can define a reduction condition that allows to remove a basis vector if it is fulfilled:

Definition 4.2. We say that a basis vector $v \in \mathcal{V}$ of the affine space V satisfies the **Reduction Condition 1** if and only if there is a column c that has an *isolated active bit* in v while $act_{U'}(c) = 0$ and $act_{\overline{v}}(c) = 0$.

This condition being fulfilled means that every vector in U' will have the bit in question inactive. If, however, the specific $v \in \mathcal{V}$ is used for the construction of an element in V this bit will always be active according to the following lemma:

Lemma 4.3. *If $b \in \mathcal{B}$ has an isolated active bit in position (c, i), then any vector in the affine space with offset b and basis $\mathcal{B} \backslash \{b\}$ has the corresponding column c activated.*

Proof. If (c, i) is an isolated active bit in $b \in \mathcal{B}$ then it is, by definition, inactive in all other basis vectors. Therefore, any vector from the described affine space has the bit activated due to b being the offset. And since it is inactive in all the other basis vectors, this bit will stay active no matter which other basis vectors are added. So, position (c, i) and also the column c will be active. □

Consequently, this specific $v \in \mathcal{V}$ cannot contribute to the construction of a vector that would ever be compatible with any $u' \in U'$ through χ. Therefore, it can be eliminated from the basis without negative effects.

This process can be efficiently done using triangulation: The bit positions in basis $\mathcal{U}'$ of the vector space U' are reordered so that the inactive bits from U' are at low indices; $\mathcal{V}, \overline{v}$ are permuted the same way (top row in Figure 7). Next, the basis $\mathcal{V}$ is triangularized to reduced row echelon form. We note the bit positions of the pivot elements during the triangulization process as these are isolated active bits in $\mathcal{V}$. At the same time $\overline{v}$ is triangularized as well, leading to a sparse $\overline{v}$ at low indices (bottom row in Figure 7). Lastly, for each pivot element the corresponding activity in U' is checked to see whether the Reduction Condition 1 is fulfilled.

Note that a case with $act_{U'}(c, i) = 0 \wedge act_V(c, i) = 1 \wedge \overline{v}(c, i) = 1$ will not occur anymore thanks to the triangularization of $\overline{v}$. This means, after triangularization, all cases with

$$act_{U'}(c) = 0 \wedge act_V(c, i) = 1$$

fulfil the Reduction Condition 1 and can therefore be removed from $\mathcal{V}$. Finally, the reordering of bit positions from the beginning is reversed, thus restoring the initial order again.

Reduction Condition 1 was phrased to target potential eliminations in the basis $\mathcal{V}$. We can, however, rephrase it to allow for eliminations in $\mathcal{U}'$:

Definition 4.4. We say that a basis vector of the vector space $u' \in \mathcal{U}'$ satisfies the **Reduction Condition 2** if and only if there is a column c that has an *isolated active bit* in u' while $act_V(c) = 0$ and $act_{\overline{v}}(c) = 0$.

Here, every $v \in V$ is inactive in the column c, while every element of U' that uses the basis vector u' would be active in it. This again follows from Lemma 4.3. So if Reduction Condition 2 is fulfilled, the specific $u' \in U'$ can be eliminated. The process of doing this efficiently is the same as for Reduction Condition 1 with the roles of U' and V being reversed.

This *Step Two* was only based on the fact that activity patterns are invariant under the S-Box layer. Since this is true for every permutation with an S-Box layer, this is a reduction condition that can be used in those cases as well.

Step Three - Columns with a single active bit: This step more actively includes the behaviour of the actual S-Box mapping used in XOODOO , which achieves further basis eliminations.

Definition 4.5. A column c of a vector b in a basis $\mathcal{B}$ is called **single bit column** if there is an i such that (c, i) is a single active bit in b and $act_{\mathcal{B}}(c, j) = 0 \ \forall \ j \in \{0, 1, 2\} \backslash \{i\}$.

Lemma 4.6. *If a state has a single bit column c with isolated bit (c, i), a compatible state through the non-linear layer χ must have an active bit at (c, i) as well.*

Proof. Assume, wlog, that the first bit in the column is active. Since the other bits are inactive, the column will follow the first row of Table 4. All of those have the first bit active. The same is true for the second or third bit. □

Definition 4.7. We say that a basis vector $v \in \mathcal{V}$ fulfils the **Reduction Condition 3** if and only if v has a *single bit column* c with single active bit at position (c, i) and $act_{\mathcal{U}'}(c, i) = 0$ and $act_{\overline{v}}(c, i) = 0$.

If this condition is fulfilled, every element in V that has the specific v as part of its linear combination will be active in this isolated bit (c, i) (Lemma 4.3). Further, Lemma 4.6 tells us that the bit position (c, i) also has to be active in any vector that matches through the non-linear layer. As a consequence, this specific basis vector $v \in \mathcal{V}$ can be removed without negative effects.

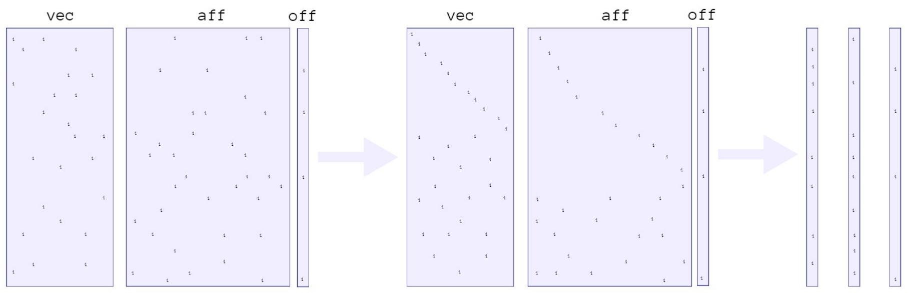


Figure 8: Schematic Overview of Step Three of Thinning Out:
'vec' is the basis $\mathcal{U}'$ of vector space U', 'aff' the basis $\mathcal{V}$ of the affine space V and 'off' the corresponding offset $\overline{v}$. First, both bases are triangularized then their activity is calculated.

To do this efficiently, both bases are first triangularized while also eliminating bits from $\overline{v}$. Both $act_{\mathcal{U}'}$ and $act_{\mathcal{V}}$ are calculated. Each column in $\mathcal{U}'$ is checked to see whether or not the Reduction Condition 3 is fulfilled. If that is the case, the basis vector is removed. This process is depicted in Figure 8.

Reduction Condition 3 was phrased to target potential eliminations in the basis $\mathcal{V}$. We can, however, rephrase it to allow for eliminations in $\mathcal{U}'$:

Definition 4.8. We say that a basis vector $u' \in \mathcal{U}'$ fulfils the **Reduction Condition 4** if and only if u' has a *single bit column* c with single active bit at position (c, i) and $act_{\mathcal{V}}(c, i) = 0$ and $act_{\overline{v}}(c, i) = 0$.

The reasoning for why this allows for a valid elimination is the same as for Reduction Condition 3. The process of actually checking for this is also the same, but with the roles of $\mathcal{V}$ and $\mathcal{U}'$ reversed.

Repeat: If, after these steps, at least one of the bases is shorter than before, it means a reduction has taken place and all steps are executed again, until no further basis elements can be eliminated. In this case, the last part of the analysis will take place. If, at any time during the process, one of the bases reaches a length of zero, it is automatically stopped. If $|\mathcal{U}'| = 0$, the next $w' \in W'$ can be chosen, since this means that $U' = \{0\}$, which is not compatible with the activity pattern. However, $|\mathcal{V}| = 0$ still leaves $V = \{\overline{v}\}$, meaning we enter a special case that will be described now.

4.3 Special Case: Full Basis Reduction of Affine Space

While developing and running the code, we realized that it was very common to end up in the following situation after the Thinning Out: The basis $\mathcal{U}'$ was reduced only by a few elements while the basis $\mathcal{V}$ was reduced completely. Therefore, we get the special case were we have to see whether any element in U' is compatible with the offset $\overline{v}$.

This is done with some similarities to *Step Two* and *Step Three* of the Thinning Out. $\mathcal{U}'$ is triangularized according to the inactive parts of $\overline{v}$. First, if any of the basis vectors in $\mathcal{U}'$ have an *isolated bit* in a column c with $\overline{v}(c) = 0$, they can be eliminated. Using it would cause the bit and therefore the column to be active in the created $u \in U$. Since it is inactive in $\overline{v}$ however, they could never be compatible (*Step Two*). Secondly, if there are any *single bit columns* in one of the basis vectors, the offset is also checked. Like explained in *Step Three*, the corresponding bit in $\overline{v}$ has to be active as they cannot be compatible otherwise. If the the offset is inactive the corresponding basis vector is removed (*Step Three*).

Both of these steps reduce $\mathcal{U}'$ quite significantly. So much so, that doing the check from *Step One* makes sense again. Since there are less vectors in $\mathcal{U}'$, there is also overall less activity. Therefore, that in many cases, a column c exists with $act_{U'}(c) = 0 \wedge \overline{v}(c) = 1$. Therefore, no compatible pairs can exist in this case!

Overall, this extra step was very helpful in the specific case we looked at, as it drastically reduced the cases that needed to be checked with the method explained in the upcoming section.

4.4 Finding Compatible Vectors

After the completed basis elimination all pairs $(u', v) \in U' \times V$ now have to be checked whether or not they are compatible through the non-linear layer χ. The easiest and least efficient way of doing this is to brute force every possible pairing.

An improvement is made like this: It is easier to look at both U' and V as affine spaces - U' has the offset zero for now. From each of the bases $\mathcal{U}'$ and $\mathcal{V}$ a basis vector b is selected with a single bit column at the same index (c, i) for both of them. For now, the offsets are required to be inactive in this column. The corresponding spaces are then split up into U_0'/U_1' and V_0/V_1 with

$$\begin{aligned} \mathcal{U}_0' &= \mathcal{U}' \backslash \{b\}, & \overline{u'}_0 &= u' \\ \mathcal{U}_1' &= \mathcal{U}' \backslash \{b\}, & \overline{u'}_1 &= u' + b \\ \mathcal{V}_0 &= \mathcal{V} \backslash \{b\}, & \overline{v}_0 &= v \\ \mathcal{V}_1 &= \mathcal{V} \backslash \{b\}, & \overline{v}_1 &= v + b \end{aligned}$$

This leads to every vector in $\mathcal{U}_0'$ and $\mathcal{V}_0$ being inactive in the corresponding bit of the Single Bit column and every vector in $\mathcal{U}_1'$ and $\mathcal{V}_1$ being active. Therefore, no compatible pairs exist between $\mathcal{U}_0'$ and $\mathcal{V}_1$ or between $\mathcal{U}_1'$ and $\mathcal{V}_0$ and these do not need to be checked. This reduces the number of comparisons needed from $2^{dim(U')+dim(V)}$ down to $2 \cdot 2^{dim(U')-1+dim(V)-1} = 2^{dim(U')+dim(V)-1}$, aka by a factor of 2. This is done recursively, multiple times to reduce all individual affine spaces as much as possible, speeding up the process significantly. Since the splitting etc. adds computations it did not make sense to reduce until the affine spaces had dimension 1. Testing showed that a reduction down to one of them reaching dimension 3 was optimal. Therefore, instead of doing further elimination, it was computationally faster to just do the checks and get the result that way.

Above it is required that the offsets $\overline{u'}$ and $\overline{v}$ have to be inactive in the picked column c as well. However, it is sometimes possible to use this splitting-up technique if there is activity in the offsets. That is, if an offset is active in said column and by adding b to the basis vector the column would become inactive. For the new offsets b is added to $\overline{u'}_0, \overline{v}_0$ instead of to $\overline{u'}_1, \overline{v}_1$. This way, the same result of one of the new affine spaces being always active and the other being always inactive is achieved!

If any valid pair is found, it still needs to be checked that the corresponding u' is actually valid, as it was picked from the overestimating U'. This is done by comparing the activity of $u = \lambda^{-1}(u')$ to that of the difference a_1 of the input trail core.

4.5 Reverse Option

Instead of iterating over $W' = \mathcal{S}_{act}(b_3)$, constructing V/V' from it and comparing that to $U' = \lambda(\mathcal{V}_{actM}(a_1))$, we can also do it in reverse order:

Construct U from the activity of a_1 via $U = \mathcal{S}_{act}(a_1)$. Use b_3 to create $W' = \mathcal{V}_{actM}(b_3)$ and obtain via $\mathcal{W} = \lambda^{-1}(\mathcal{W}')$ the basis for W. Iterate over all elements from U: Compute $u' = \lambda(u)$, then create an affine space V from u and use the linear layer to get a corresponding affine space V'. Thin out V' and W against each other. Then check the last pairs $(v', w) \in V' \times W$ to find any potential compatible pairs. If any were found, check activity again before outputting. This is also described in Algorithm 5.

This reversed version does the same as the classical one, with the only difference that we iterate over U instead of W'. For ease of reading, the main text only references the classical version at all times. But note that everything still works the same, the creation of an affine space, the thinning out and the search for compatible pairs of vectors. This works, because the S-Box is symmetric and compatible pairs create an affine space in both directions as mentioned in Section 3.4.

The reason for using this reverse option would be an improvement in run time. The actions for each element of the activity matching set (creating an affine space, thinning it out against the vector space, looking for compatible pairs) have either very low run times or unpredictable ones: Constructing the affine space always runs in short, constant time as it iterates over all columns c. The thinning out cannot be predicted properly and the same goes for finding compatible pair as it depends on the results of the thinning out.

Therefore, the run time is mainly correlated to the size of the activity matching set. This is useful in the case of a trail where $\#_{act}(a_1)$ is lower than $\#_{act}(b_3)$, since the size is directly correlated to $|\mathcal{S}_{act}(a_1)|$. If the reverse is true, it is faster to turn around the entire algorithm as described above with a run time now correlated to the lower $|\mathcal{S}_{act}(b_3)|$.

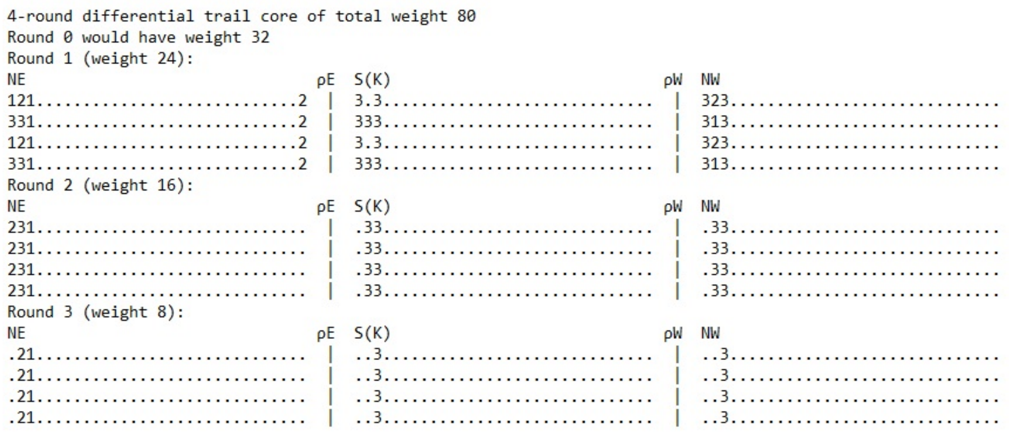
Algorithm 5 Pseudo Code for 4 Round Analysis - Reverse Version

```
Require: $a_1, b_3$
  $W' \leftarrow \mathcal{V}_{actM}(b_3)$
  $\mathcal{W} \leftarrow \lambda^{-1}(\mathcal{W}')$
  $U \leftarrow \mathcal{S}_{act}(a_1)$
  for $u \in U$ do
    $u' \leftarrow \lambda(u)$
    $V \leftarrow$ affineSpace($u'$)
    $\mathcal{V}' \leftarrow \lambda^{-1}(\mathcal{V}')$
    $(\mathcal{V}', \mathcal{W}) \leftarrow$ thinOut($\mathcal{V}'$, $\mathcal{W}$)
    for $v' \in V'$ and $w \in W$ do
      if $v'$ and $w$ are compatible then
        $w' \leftarrow \lambda(w)$
        if $act_{w'} = act_{b_3}$ then
          output $(u, \lambda^{-1}(v'), w)$
        end if
      end if
```

5 Results of the Analysis

This section shows the trail core we used exemplary for our analysis in this thesis. It also gives the result of said analysis.

5.1 Used Trail Core



```
4-round differential trail core of total weight 80
Round 0 would have weight 32
Round 1 (weight 24):
NE                              ρE S(K)                            ρW NW
121............................2 | 3.3............................. | 323.............................
331............................2 | 333............................. | 313.............................
121............................2 | 3.3............................. | 323.............................
331............................2 | 333............................. | 313.............................
Round 2 (weight 16):
NE                              ρE S(K)                            ρW NW
231............................. | .33............................. | .33.............................
231............................. | .33............................. | .33.............................
231............................. | .33............................. | .33.............................
231............................. | .33............................. | .33.............................
Round 3 (weight 8):
NE                              ρE S(K)                            ρW NW
.21............................. | ..3............................. | ..3.............................
.21............................. | ..3............................. | ..3.............................
.21............................. | ..3............................. | ..3.............................
.21............................. | ..3............................. | ..3.............................
```

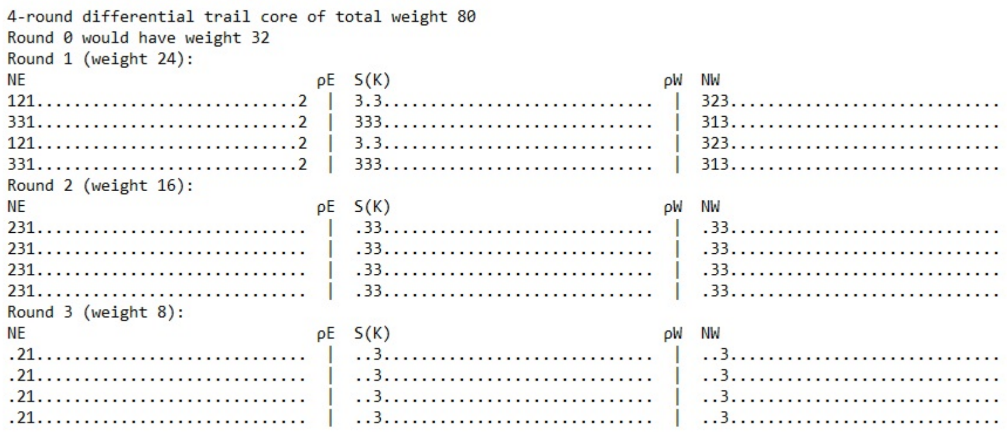
Figure 9: Trail core that was used for search of clustering cores. It was found by Daemen et al. [5]. This picture is a screenshot of the output of their implementation.

Figure 9 shows the trail core used exemplary throughout this thesis. Round 0 has been omitted as we focus on the trail core only. Rounds $1, 2$ and 3 are shown individually in consecutive lines. 'pE' and 'pW' indicate ρ_{east} and ρ_{west} respectively. The non-linear layer χ occurs at the end of each round, aka after the last state shown in a line and before the first state in the next one. In this specific case, applying the function θ that is part of the linear layer is not shown. It would occur between ρ_{east} and ρ_{west}. However, it does not alter the state, which means it would be written twice. To keep this narrower, it was therefore left out. Instead of showing each bit individually, the numbers represent column values in bird's eye perspective. Therefore, a 1 means this bit position has a 1 in the plane with index $y = 0$, while planes at $y = 1, y = 2$ have a bit value 0.

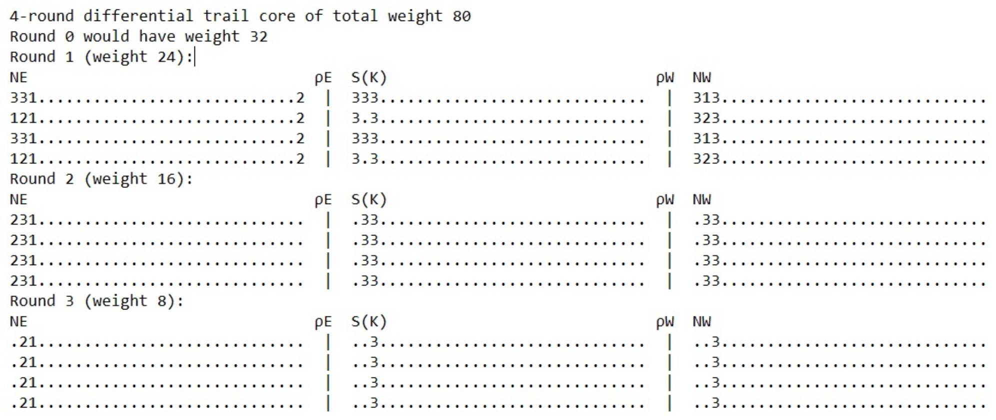
5.2 Results and Observations

We implemented the approach presented in this thesis. It was written in Python as the 3-round version was as well and this allowed us to take over a few parts. The *regular* was implemented, the *reverse* version was not but mainly requires an inversion of order in regards to the core components of the program. The implementation as well as documentation on it can be found here [https://github.com/CDworzack/4RoundAnalysis].

This code was executed using the trail core shown in Section 5.1. It was run on a laptop equipped with an Intel® Core^{TM} i5-10210U CPU. The run time was around 1 hour and 40 minutes.



```
4-round differential trail core of total weight 80
Round 0 would have weight 32
Round 1 (weight 24):
NE                              ρE  S(K)                            ρW  NW
121............................2 |  3.3.............................  |  323.............................
331............................2 |  333.............................  |  313.............................
121............................2 |  3.3.............................  |  323.............................
331............................2 |  333.............................  |  313.............................
Round 2 (weight 16):
NE                              ρE  S(K)                            ρW  NW
231.............................  |  .33.............................  |  .33.............................
231.............................  |  .33.............................  |  .33.............................
231.............................  |  .33.............................  |  .33.............................
231.............................  |  .33.............................  |  .33.............................
Round 3 (weight 8):
NE                              ρE  S(K)                            ρW  NW
.21.............................  |  ..3.............................  |  ..3.............................
.21.............................  |  ..3.............................  |  ..3.............................
.21.............................  |  ..3.............................  |  ..3.............................
.21.............................  |  ..3.............................  |  ..3.............................
```



```
4-round differential trail core of total weight 80
Round 0 would have weight 32
Round 1 (weight 24):
NE                              ρE  S(K)                            ρW  NW
331............................2 |  333.............................  |  313.............................
121............................2 |  3.3.............................  |  323.............................
331............................2 |  333.............................  |  313.............................
121............................2 |  3.3.............................  |  323.............................
Round 2 (weight 16):
NE                              ρE  S(K)                            ρW  NW
231.............................  |  .33.............................  |  .33.............................
231.............................  |  .33.............................  |  .33.............................
231.............................  |  .33.............................  |  .33.............................
231.............................  |  .33.............................  |  .33.............................
Round 3 (weight 8):
NE                              ρE  S(K)                            ρW  NW
.21.............................  |  ..3.............................  |  ..3.............................
.21.............................  |  ..3.............................  |  ..3.............................
.21.............................  |  ..3.............................  |  ..3.............................
.21.............................  |  ..3.............................  |  ..3.............................
```

Figure 10: On top the original trail core used as input to the implementation. On the bottom the other trail that was found which clusters with the first one.

In total, two trail cores were found by the code. One of them is the original one described in Section 5.1. The other one was very similar to the original and they are both displayed in Figure 10. The only difference is that within Round 1, the 2nd and 4th line seem to be switched up with the 1st and 3rd. The Rounds 2 and 3 are completely identical.

All other instances of $w' \in W'$ ended up being reduced completely so that the code never reached the steps described in Section 4.4. As can be seen from Figure 10, Round 3 is identical which means that they both resulted from the same $w' \in W'$.

Out of the $7^4 = 2401$ $w' \in W'$ that were checked, in around 100 cases, $\mathcal{U}'$ was completely eliminated thanks to *Step One*. All other cases ended up in the *Special*

Case described in Section 4.3. This special case then managed to again fully eliminate the basis $\mathcal{U}'$. Therefore, the divide-and-conquer technique was never actually used. In the case that actually needed to be checked $\mathcal{V}$ was reduced all the way, such that $dim(V) = 0$. Therefore, the remaining elements from U were just checked one by one against the offset $\overline{v}$.

6 Conclusion

In this thesis we first explained a general approach on how to find clustering trail cores to a given trail core (Section 4). We also wanted to find clustering trail cores to a specific 4-round trail core for Xoodoo . Thereby, we were going to answer the research question given at the beginning. Yes, it is possible to efficiently detect the existence of clustering trails and find them.

The idea was implemented and then tried which resulted in finding one clustering trail as explained in Section 5.2. It also turned out that one of the concepts thought up for this search was not used during the running of the algorithm, namely the last step explained in Section 4.4. Further tests on other trails would reveal how common this occurrence is. The reverse version has also not been used at this point.

As said in Section 4.5 the run time of the code mainly depends on $\mathrm{w}(b_3)$ or $\mathrm{w}(a_1)$ depending on the direction. Many cases exist that have higher weights at those positions, optimizations to the code or the overall concept would make the search for clustering cores on those more feasible.

A Appendix

The function *affineColumn* is described by the following Algorithm:

Algorithm 6 affineColumns:
Returns the offset and both base vectors for the affine space that describe the compatible columns given an input column.

```
Require: $v(c)$
  if $v(c) = (0,0,1)$ then
      return $(0,0,1), (0,1,0), (1,0,0)$
  end if
  if $v(c) = (0,1,0)$ then
      return $(0,1,0), (0,0,1), (1,0,0)$
  end if
  if $v(c) = (1,0,0)$ then
      return $(1,0,0), (0,0,1), (0,1,0)$
  end if
  if $v(c) = (0,1,1)$ then
      return $(0,1,0), (1,0,0), (0,1,1)$
  end if
  if $v(c) = (1,0,1)$ then
      return $(0,0,1), (0,1,0), (1,0,1)$
  end if
  if $v(c) = (1,1,0)$ then
      return $(1,0,0), (0,0,1), (1,1,0)$
  end if
  if $v(c) = (1,1,1)$ then
      return $(1,0,0), (0,1,1), (1,0,1)$
```

References

[1] Eli Biham and Adi Shamir. Differential cryptanalysis of DES-like cryptosystems. Journal of Cryptology, 1991. https://doi.org/10.1007/BF00630563.

[2] Nicolas Bordes, Joan Daemen, Daniël Kuijsters, and Gilles Van Assche. Thinking outside the superbox. Cryptology ePrint Archive, Paper 2021/293, 2021.

[3] Joan Daemen, Seth Hoffert, Gilles Van Assche, and Ronny Van Keer. The design of Xoodoo and Xoofff. *IACR Transactions on Symmetric Cryptology*, 2018.

[4] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Information Security and Cryptography. Springer, 2002.

[5] Gilles Van Assche Joan Daemen, Silvia Mella. Tighter trail bounds for xoodoo. Cryptology ePrint Archive, Paper 2022/1088, 2022. https://eprint.iacr.org/2022/1088.