RUB

RUHR-UNIVERSITÄT BOCHUM

Differential Trail Weights in AES-like Ciphers Using New Permutation Layers

Christof Beierle

Master's Thesis. September 18, 2014.
Chair for Embedded Security – Prof. Dr.-Ing. Christof Paar
Advisor: Dr. Gregor Leander

hgi EMSEC

Abstract

Differential cryptanalysis is about finding trails which involve a low amount of active S-box operations. Such patterns simplify the difference propagation and thus the computations of the secret round keys. The weight of a trail expresses the total number of non-zero S-box input differences within the cipher for a certain execution and is highly dependent on the specified permutation layers. This work copes with analyzing possible permutations in AES-like structures in order to guarantee high-weight trails. The main focus lies on cubed state dimensions. After providing a formalization on the concept of guaranteed trail weights, some characteristcs of equivalent permutation layers due to this definition are shown. One of the advantages is that the definition of guaranteed active S-boxes is made independently of the concrete mixings and the used S-box function. In the experimental part of this thesis, lots of possibilities are tested, using a mixed-integer linear programming approach. A certain focus lies on cyclic rotations within the rows, a generalization of the AES `ShiftRows` permutation, since they are rather easy to implement. As one result it turns out that an eight-round trail weight of B_θ^3 can be reached for cubed dimensions using these cyclic rotations. Thereby, B_θ denotes the branch number of the linear mixing step. This is also proven using an iteration of the AES Four-Round Propagation Theorem.

Statutory Declaration

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of the university or other institute of higher learning, except where due acknowledgment has been made in the text.

Eidesstattliche Erklärung

Ich erkläre, dass ich keine Arbeit in gleicher oder ähnlicher Fassung bereits für eine andere Prüfung an der Ruhr-Universität Bochum oder einer anderen Hochschule eingereicht habe.

Ich versichere, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Die Stellen, die anderen Quellen dem Wortlaut oder dem Sinn nach entnommen sind, habe ich unter Angabe der Quellen kenntlich gemacht. Dies gilt sinngemäß auch für verwendete Zeichnungen, Skizzen, bildliche Darstellungen und dergleichen.

Christof Beierle

Contents

1 Introduction **1**
1.1 Motivation 1
1.2 Related Work 2
1.3 Organization of this Thesis 2

2 The Advanced Encryption Standard and AES-like Ciphers **3**
2.1 The Selection Process 3
2.2 Structure of the Cipher 3
2.2.1 The State 4
2.2.2 The SubBytes Layer 4
2.2.3 The ShiftRows Layer 5
2.2.4 The MixColumns Layer 5
2.2.5 The Round Key Addition 6
2.3 AES-like Ciphers 6
2.3.1 Arranging the State 7

3 Differential Trails and Guaranteed Trail Weights **9**
3.1 The Idea behind Differential Attacks 9
3.2 Active S-Boxes and Branch Numbers 10
3.2.1 Optimal Branch Numbers from MDS Codes 11
3.3 Guaranteed Active S-Boxes as a Lower Bound 12
3.4 Characterizing Permutation Sequences 14
3.4.1 A Standard Form for Describing Permutation Layers 18

4 Computing Guaranteed Trail Weights **21**
4.1 Mixed-Integer Linear Programming 21
4.2 The Cipher as a Mixed-Integer Linear Programming Model 22
4.3 A First Approach on the Minimization Problem 25
4.3.1 Dimension 2 x 4 27
4.3.2 Dimension 3 x 9 27
4.3.3 Dimension 4 x 16 27
4.4 Finding Optimal Permutations 29
4.4.1 Considering Arbitrary Permutations 30
4.4.2 Considering Two-Alternating Cyclic Rotations 30
4.4.3 Reaching the Optimum with Cyclic Rotations 33

5 Round Propagation **35**
5.1 The Standard AES Case . . . 35
5.2 Superboxes . . . 35
5.3 Eight-Round Propagation with Iterated Superboxes . . . 36
5.4 Disadvantages . . . 38

6 Conclusion **39**
6.1 Summary . . . 39
6.2 Future Work . . . 39

A Listings of the MILP Tools **41**

List of Figures **57**

List of Tables **59**

Bibliography **61**

1 Introduction

Cryptography plays an important role in nearly all kind of today's electronic devices. In order to maintain secure communication and the protection of sensitive data, the way of designing strong and efficient cryptography is and remains an important research area nowadays. Especially block ciphers are one of the most important primitives since they are usually realizable in an efficient way, either in hardware or in software.

1.1 Motivation

Today's state-of-the-art block ciphers are usually designed as a substitution-permutation network (SPN), mixing the principles of diffusion and confusion, which go back to C. Shannon [Sha49]. Especially AES-like structures may likely occur in symmetric cryptography as a realization of SPNs (LED [GPPR11], ECHO [BBG$^+$09], Grøstl [GKM$^+$08],etc.). As the last two examples express, the AES-based structure was a popular suggestion for some of the SHA-3 candidates. Thus, the results developed within this work can also be applied to hash functions.

Although there exist several works on designing the non-linear layer, there is less focus on the permutation part. It is not that easy to cope with the analysis of trail weights since they are dependent on the S-boxes and the linear mixing steps. In addition, the trail pattern from a certain round on is dependent on the outcome of the previous rounds, resulting in a rather not well-understood structure. Moreover, considering arbitrary permutations on different dimensions contains combinatorial difficulties as well.

One goal of this thesis is to develop a structured approach on these difficulites in a more formalized way. Considering guaranteed trail weights as a lower bound, instead of actual weights, simplifies the problem by making it independent of the non-linear layer and the concrete mixing steps. Using mixed-integer linear programming, this lower bound can easily be computed. The intention is to establish a better understanding of differential trails and this work should be a first step in analyzing them due to permutation layers. With the AES and its standard squared state dimension of 4×4, a strong permutation layer is already defined with the `ShiftRows` operation. However, the interest may also be in understanding other non-squared states, especially cubed dimensions. Using such states would be interesting for more lightweight approaches such as bit slicing. Thus, the focus in this thesis lies on these cubed dimensions. Finding an optimal choice for the permutation layer becomes not that trivial anymore. In addition, since cyclic rotations within the rows are rather good to implement, one would appreciate to use these kind of permutations within the cipher. Thus, analyzing AES variants in different dimensions is worth the effort in order to deepen the understanding of these structures.

As already stated above, the concepts are not restricted to block cipher design. Instead, the knowledge of permutation layers may also be useful in order to design secure hash functions.

1.2 Related Work

There are several works on using mixed-integer linear programming for symmetric cryptanalysis. [MWGP11] describes how to use this method in order to determine a lower bound on the trail weight in the Advanced Encryption Standard and provides a verification of the Four-Round Propagation Theorem [DR02, p. 140]. The concepts presented in this thesis are based on these ideas.

A secure design of a cubed-state AES-like cipher is already described in [BK14]. The permutation layer consists of a `ShiftRows` and a shuffle step, which contains an iterated superbox structure. This work verifies the proven lower bound of 125 active S-boxes over eight rounds and shows how this bound can also be reached using only cyclic rotations within the rows as the permutation layer.

1.3 Organization of this Thesis

Chapter 2 briefly introduces the structure of the Advanced Encryption Standard and its variants which will be part of the later analysis. Chapter 3 is dealing with the concept of differential attacks and presents the mathematical preliminaries used for the experimental part. Furthermore, it formalizes the idea behind guaranteed active S-boxes and provides some useful theorems on this concept. Thus, beneath Chapter 4, it contains the main part of this work. The other main chapter (4) explains how the amount of guaranteed active S-boxes can easily be determined using a mixed-integer linear programming approach. Making use of these basic definitions and theorems, an experimental analysis on some possible permutation layers in the three smallest cubed dimensions is done. As a main result, a strong permutation layer is defined which only uses cyclic rotations within the rows. Finally, in Chapter 5 its security is proven for arbitrary cubed dimensions using an iterated superbox argument.

2 The Advanced Encryption Standard and AES-like Ciphers

The Advanced Encryption Standard (AES) is a block cipher based on a substitution-permutation network. It was first presented in 1998 under the name "Rijndael" by the Belgium scientists Joan Daemen and Vincent Rijmen as a submission for the AES selection process [DR98]. It is todays state-of-the-art and widely-used cipher in all kind of applications.

2.1 The Selection Process

In the year 1997, the National Institute for Standards and Technology (NIST) in Maryland, USA started an open competition in order to design a new block cipher to become the Advanced Encryption Standard. It was intended to be a successor of the Data Encryption Standard (DES) which already had its weaknesses according to its small key length of only 56 bit. The Rijndael cipher was one of the 15 submissions and finally chosen as the AES. [DR02]. The official specification became a Federal Information Processing Standard (FIPS) and can be found in [FIP01].

Since the only difference between Rijndael and the later AES lies in the supported block length, the two will be used as synonym.[1]

2.2 Structure of the Cipher

In order to cope with differential attacks on Rijndael and Rijndael-based ciphers in the remainder of this thesis, a brief overview on the stucture of the cipher will be given. More in-depth details can be found in [PP10] or [DR02]. The notation used in these sections is based on the literature. However, the state will be denoted with two-dimensional indices due to better compatibility and understanding resasons in the next chapters.

The AES operates on a block length of 128 bits and supports the three key length 128, 192 and 256 bit. It is a key-iterated block cipher, which means that a certain transformation (called round transformation) is iterated over several rounds. Every round consists of a different round key which is derived from the cipher key. Depending on the key length, there will be a different number of rounds as shown in table 2.1.

The round transformation is defined by substitution and permutation steps followed by a round key addition. Precisely, one round consists of the steps `SubBytes`, `ShiftRows`,

[1] For simplicity, only the block length of 128 bit (AES) will be considered. However, it should be easy to adapt the concept to different block length.

Table 2.1: Number of rounds for certain key length [PP10].

key length	rounds
128 bit	10
192 bit	12
256 bit	14

MixColumns and AddRoundKey transforming the actual state[2] as

$$\texttt{AddRoundKey}_{K_r} \circ \texttt{MixColumns} \circ \texttt{ShiftRows} \circ \texttt{SubBytes}\,.$$

2.2.1 The State

The plaintext of a certain block is arranged in a specific order, called the state. Each layer in each round transforms the state into a new intermediate state. The 128 bits are grouped in a bytewise manner and are wirtten in a 4x4 matrix.

$a_{(0,0)}$	$a_{(0,1)}$	$a_{(0,2)}$	$a_{(0,3)}$
$a_{(1,0)}$	$a_{(1,1)}$	$a_{(1,2)}$	$a_{(1,3)}$
$a_{(2,0)}$	$a_{(2,1)}$	$a_{(2,2)}$	$a_{(2,3)}$
$a_{(3,0)}$	$a_{(3,1)}$	$a_{(3,2)}$	$a_{(3,3)}$

Figure 2.1: Illustration of the AES state.

The bundles $a_{(m,n)}$ consist of 8 bits each in the case of the standard AES case. In general, the bit length of each bundle is given by l. Thus, each of them can be considered as an element of the Galois field $GF(2^l)$. If one wants to consider other block lengths, the state would just be arranged in another dimension, for example 4×6 in case of 192 bits. Since the MixColumns layer uses a linear mapping over $GF(2^8)^4$, finite field arithmetic needs to be done. For that, the standardized irreducible reduction polynomial $x^8 + x^4 + x^3 + x + 1$ is used.

2.2.2 The SubBytes Layer

The SubBytes layer operates on each bundle seperately. It substitutes the content of a bundle by another value described by the AES S-box which is the only non-linear transformation with respect to the XOR-operation. In gereral, the S-box is defined as a non-linear bijective mapping

$$GF(2^l) \to GF(2^l), \quad a_{(m,n)} \mapsto \texttt{S}(a_{(m,n)}),$$

in which l denotes the bit length. The non-linearity is referred to the bitwise XOR, which means that in general

$$\texttt{S}(a_{(m_1,n_1)} \oplus a_{(m_2,n_2)}) \neq \texttt{S}(a_{(m_1,n_1)}) \oplus \texttt{S}(a_{(m_2,n_2)})$$

[2]The last round differs from the others as there is no MixColumns layer. This will not cause any problems in the analysis later.

holds for at least one pair $a_{(m_1,n_1)}, a_{(m_2,n_2)}$. In order to prevent vulerabilities of the cipher, the S-box `S` must be well chosen and the non-linearity is a necessary condition on security.

The AES S-box is defined by a composition of a multiplicative invertion over $GF(2^8)$ and a bijective affine mapping over this field. Since the cipher has no Feistel structure, it is important that all transformations are injective. Since the secure S-box design is not the topic of this thesis, it is not descussed within this work.

2.2.3 The ShiftRows Layer

The `ShiftRows` step is a linear transformation permuting the bundles within the state. As the name already suggests, it just shifts the four rows of the state by different offsets. The first row is unchanged, while the second, third and fourth is left-rotated by one, two and three steps. In mathematical sense, the permutation step can be described by a permutation π over the index space. Formally, let $S_{M\times N}$ denote the set of all permutations on the cartesian product $M \times N$. Thus, every permutation $\pi \in S_{M\times N}$ can be considered as $\pi(m,n) = (\pi^\alpha(m,n), \pi^\beta(m,n))$. Thereby, $\pi^\alpha(m,n)$ denotes the row and $\pi^\beta(m,n)$ the column of the new state bundle which will be at position (m,n) after that transposition. In case of the `ShiftRows` step in the original 4×4 state that would be

$$\begin{aligned}\pi : \{0,1,2,3\} \times \{0,1,2,3\} &\to \{0,1,2,3\} \times \{0,1,2,3\}\\ (m,n) &\mapsto (m, n+m \mod 4).\end{aligned}$$

Considering generalized AES-like ciphers, one would allow arbitrary bundle permutations within the state, which can be different depending on the current round. In the remainder of this thesis, the permutation step is always denoted by $\texttt{Permute}_\pi$ for a given $\pi \in S_{M\times N}$.

2.2.4 The MixColumns Layer

This is the next transformation of the linear layer, which is a mixing step operating on the columns. It is the most complex step with respect to the computational effort and the running time. If $(a_{(0,n)}, a_{(1,n)}, a_{(2,n)}, a_{(3,n)})$ denotes the n-th column of the AES input state, the output of the `MixColumns` operation can be described as

$$\begin{pmatrix} a'_{(0,n)} \\ a'_{(1,n)} \\ a'_{(2,n)} \\ a'_{(3,n)} \end{pmatrix} = \begin{pmatrix} \texttt{0x02} & \texttt{0x03} & \texttt{0x01} & \texttt{0x01} \\ \texttt{0x01} & \texttt{0x02} & \texttt{0x03} & \texttt{0x01} \\ \texttt{0x01} & \texttt{0x01} & \texttt{0x02} & \texttt{0x03} \\ \texttt{0x03} & \texttt{0x01} & \texttt{0x01} & \texttt{0x02} \end{pmatrix} \cdot \begin{pmatrix} a_{(0,n)} \\ a_{(1,n)} \\ a_{(2,n)} \\ a_{(3,n)} \end{pmatrix}$$

The matrix entries are constants over $GF(2^8)$ represented by their hexadecimal value. As necessary for decryption, the transformation is a bijection, which means that the matrix is invertible. For security terms, the matrix generates a maximum distance separable (MDS) code.

The most important design criteria of this linear mapping are implementation- and security properties. The small matrix coefficients are due to implementation advantages in 8 bit microprocessors. The `MixColumns` step is designed in a way, that it has optimal diffusion properties. This is expressed by a branch number of 5. It is a measure for the number of active bundles per column in any two consecutive rounds and is formally defined in the next chapter. There also follow more details on MDS codes.

2.2.5 The Round Key Addition

As mentioned before, every round has its own round key which is derived by a key scheduling algorithm. For simplicity, the round keys will be assumed to be uniformly distributed and independent from each other. In particular, our differential analysis is independent from the concrete keys. Thus, the algorithm will not be described here.

Let the key of round r be denoted by $K_r = (k^{(r)}_{(0,0)}, k^{(r)}_{(0,1)}, ..., k^{(r)}_{(3,3)})$. The bundle $a_{(m,n)}$ is simply added by its key byte $k^{(r)}_{(m,n)}$ completing the round. For other state dimensions, the key is correspondingly longer or shorter. Resuming all the layers, Figure 2.3 shows a general AES round.

2.3 AES-like Ciphers

The Rijndael cipher is designed in a way that provides strong resistance against differential attacks since the linear layer has optimal diffusion power and guarantees a high amount of active S-boxes. The question is what happens when the state is arranged in a different way, for example as a $4 \times 4 \times 4$ or, considering lower dimensions, as a $2 \times 2 \times 2$ cube. For generalization, the state bundles need not to contain elements of $GF(2^8)$ anymore. Instead, an arbitrary but fixed bit length l is allowed. Of course, the transformations have to be adjusted.

More formally, an AES-like cipher is defined in the following way:

Definition 2.3.1. An *AES-like cipher* $\mathfrak{C}$ consists of a state dimension $M \times N$ with M rows and N columns, a bit length l, a number of rounds R and the following bijective transformations operating on the state:

1. `SubBytes` operates on each bundle seperately and substitutes every bundle entry with its S-box value.

$$\texttt{SubBytes} : GF(2^l)^{M\times N} \to GF(2^l)^{M\times N}, \quad (a_{(m,n)})_{M\times N} \mapsto (\texttt{S}(a_{(m,n)}))_{M\times N}$$

2. $\texttt{Permute}_{\pi_r}$ permutes the bundles within the state due to a given permutation $\pi_r \in S_{M\times N}$ depending on the round.

$$\texttt{Permute}_{\pi_r} : GF(2^l)^{M\times N} \to GF(2^l)^{M\times N}, \quad (a_{(m,n)})_{M\times N} \mapsto (a_{\pi_r(m,n)})_{M\times N}$$

3. $\texttt{MixColumns}_{\texttt{MC}^{(r)}_1,...,\texttt{MC}^{(r)}_N}$ operates on each column of the state seperately and applies a linear mixing step to the column. These automorphisms are given by the square

matrices $\mathtt{MC}_n^{(r)} \in GF(2^l)^{M\times M}$ depending on the round r and the column n. Thus, for every round and every column, a different mixing is allowed.

$$\mathtt{MixColumns}_{\mathtt{MC}_1^{(r)},\dots,\mathtt{MC}_N^{(r)}} : GF(2^l)^{M\times N} \to GF(2^l)^{M\times N},$$
$$(col_n)_N \mapsto (\mathtt{MC}_n^{(r)} \cdot col_n)_N$$

with $col_n = (a_{(0,n)}, a_{(1,n)}, \dots, a_{(M-1,n)})$. The transformation step is also denoted by $(\mathtt{MC}_1^{(r)}, \dots, \mathtt{MC}_N^{(r)})$.

4. $\mathtt{AddRoundKey}_{K_r}$ XORes the round key K_r to the state.

Every round $r \in \{1, \dots, R\}$ within the cipher is thus executed as

$$\mathrm{rnd}_{K_r,\pi_r,\mathtt{MC}_1^{(r)},\dots,\mathtt{MC}_N^{(r)}} = \mathtt{AddRoundKey}_{K_r} \circ (\mathtt{MC}_1^{(r)}, \dots, \mathtt{MC}_N^{(r)}) \circ \mathtt{Permute}_{\pi_r} \circ \mathtt{SubBytes}$$

depending on the round key K_r, the round permutation π_r and the linear column mixings.

The AES-like cipher $\mathfrak{C}$ *uses permutations* $\pi_1, \dots, \pi_r$, if and only if these are the index permutations of the corresponding `Permute` steps.

2.3.1 Arranging the State

In this work, mainly cube-arranged states are considered. During the eperimental part, the consideration may be restricted to a small dimension due to computational issues. The bundle bit length will be arbitrary but fixed, as long as it equals at least 3 ($l \geq 3$). This restriction is due to the fact that there are no useful two-bit S-boxes. Because of the generalization on the dimensions, different cipher block length are possible. For simplicity, the state will often be represented two-dimensional. Each layer of the cube is pushed to the right side of the two-dimensional state matrix. Thus, a $2 \times 2 \times 2$ cube is represented as a 2×4 matrix and a $4 \times 4 \times 4$ cube as a 4×16 matrix. Figure 2.2 illustrates this notation.

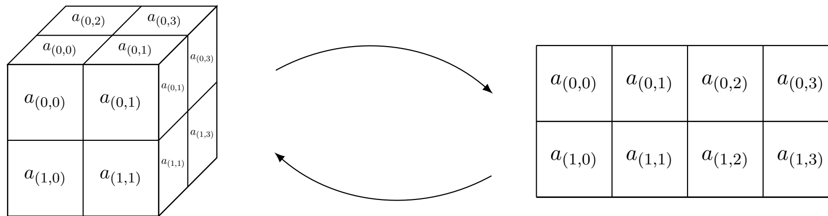

Figure 2.2: Simple interpretation of the state with dimension $2 \times 2 \times 2$.

The `MixColumns` layer should have an optimal branch number depending on the column dimensions. In addition, another S-box has to be used when the bundle bit length is not 8. The major goal is to find permutations replacing the `ShiftRows` step that have good properties on security, which means that they maximize the number of active bundles. We also allow more than one permutation which will be processed depending on the round.

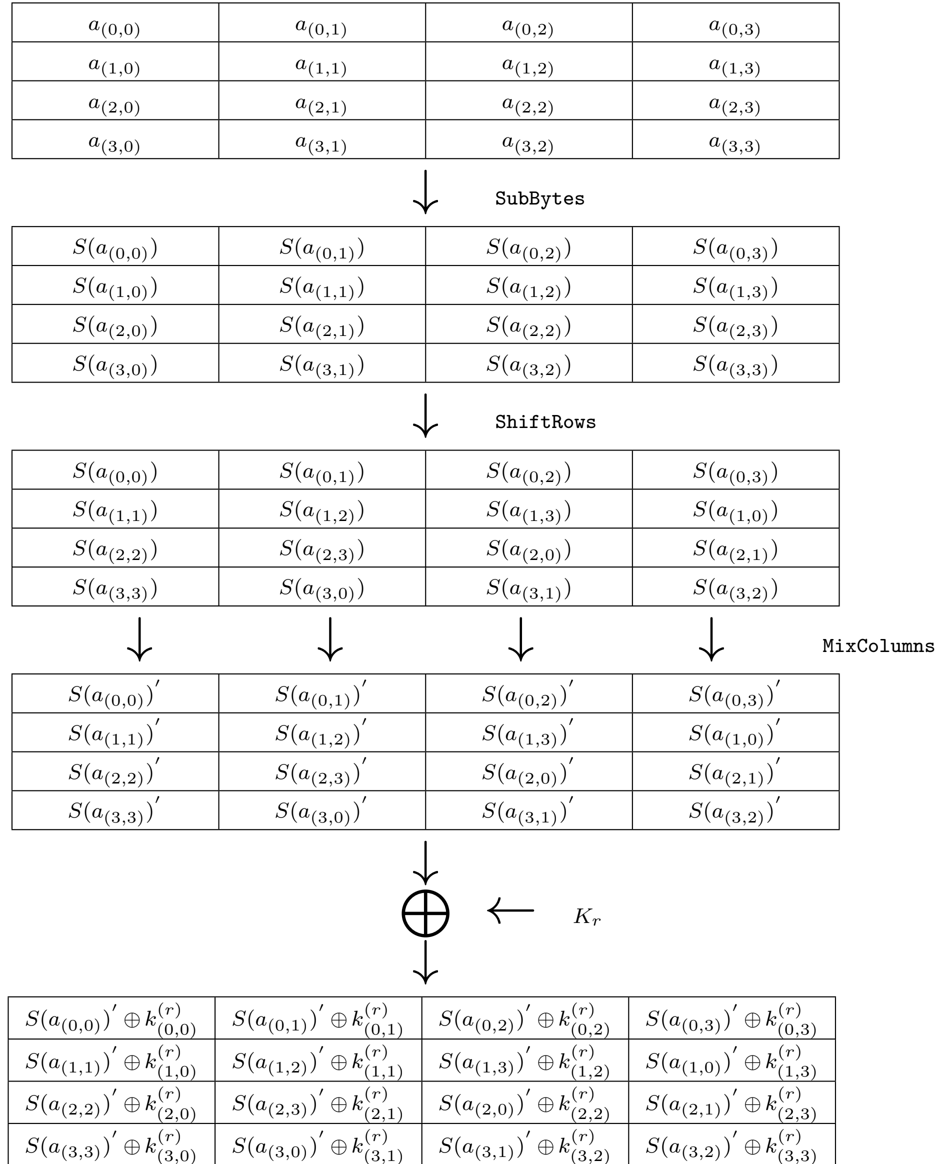

Figure 2.3: Illustration of one round r of the AES-encryption.

3 Differential Trails and Guaranteed Trail Weights

Together with linear cryptanalysis, the differential attacks are the basic kind of vulnerabilities, every new-designed block cipher should provide restistance against. The concept behind differential cryptanalysis was first published in 1991 by Eli Biham and Adi Shamir with a consideration on DES [BS91].

This chapter shortly describes the idea behind these attacks with a focus on trail weights. Section 3.3 then introduces some definitions and Section 3.4 provides properties of permutation sequences with respect to some implications on their resulting trail weights. These preliminaries can later be used in the experimental work.

3.1 The Idea behind Differential Attacks

Differential attacks are about deducing properties of the secret key considering lots of pairs of input plaintexts and output ciphertexts. Precisely, for plaintexts p_1, p_2 and ciphertexts c_1, c_2, the propagation of the XOR-difference $p_1 \oplus p_2$ resulting in $c_1 \oplus c_2$ is analyzed. The major advantage of this consideration is the fact, that the addition of a certain round key has no effect on the intermediate differences, since $(p_1 \oplus K) \oplus (p_2 \oplus K) = p_1 \oplus p_2 \oplus K \oplus K = p_1 \oplus p_2$ for any K. In general, for all linear layers within the cipher, the output difference can be calculated from the input difference and vice versa. This is due to the homomorphism property of linear maps. In order to do the difference propagation through the cipher rounds, the remaining components are the non-linear S-boxes. This uncertainty is the major reason why the substitution layer has to be non-linear in order to avoid these kind of attacks. Analyzing such non-linear maps, for each input difference I and each output difference O, the probability $I \xrightarrow{\mathsf{S}} O$ is computed. This results in a probability distribution table which describes the probabilities of input differences mapping to a certain output difference. Practical attacks have to consider multi-round differentials in order to propagate through the whole cipher. Thus, beneath a careful S-box design, there are countermeasures coping with the linear layer as well. A detailed description of differential (and linear) cryptanalysis is given in [KR11].

In order to avoid these vulnerabilities, there are three important design criteria for block ciphers.

1. **Balancing the S-box difference distribution:** There should not be any significant high probability $I \xrightarrow{\mathsf{S}} O$. An ideal approach should be an almost uniform distribution of the possible results.

2. **Designing the linear layer in order to obtain high diffusion:** Diffusion is a measure of how many of the output bits depend on all of the input bits. High diffusion leads to a harder analysis.

3. **Avoiding low-weight trails:** This is related to the diffusion approach. If there are no input plaintext differences such that only a few of the S-boxes have a non-zero input difference, the propagation will have to be done for lots of non-linear transformations at once.

This work deals with guaranteeing high-weight trails and thus has a focus on the third aspect.

3.2 Active S-Boxes and Branch Numbers

In the remainder of this thesis, an important aspect lies on the analysis of differential trails and their weights. Therefore, it is necessary to formally define them at first. Since they are one of the basic objects in differential cryptanalysis, similar definitions can be found in various literature.

Definition 3.2.1. Let a bit length l and a state dimension of $M \times N$ be given. An *R-round trail* is a sequence in $GF(2^l)^{M\times N}$ of length $R+1$:

$$GF(2^l)^{M\times N} \to GF(2^l)^{M\times N} \to \cdots \to GF(2^l)^{M\times N}.$$

For an R-round trail

$$\mathfrak{T}: \begin{pmatrix} t^{(0)}_{(0,0)} & \cdots & t^{(0)}_{(0,N-1)} \\ \vdots & \ddots & \vdots \\ t^{(0)}_{(M-1,0)} & \cdots & t^{(0)}_{(M-1,N-1)} \end{pmatrix} \longrightarrow \ldots \longrightarrow \begin{pmatrix} t^{(R)}_{(0,0)} & \cdots & t^{(R)}_{(0,N-1)} \\ \vdots & \ddots & \vdots \\ t^{(R)}_{(M-1,0)} & \cdots & t^{(R)}_{(M-1,N-1)} \end{pmatrix}$$

the *(R-round) weight of* $\mathfrak{T}$ is defined as the amount of non-zero components in the first R matrices. Formally it is

$$\text{weight}(\mathfrak{T}) := \sum_{\substack{(m,n)\in M\times N \\ 0\le r\le R-1}} \delta_{t^{(r)}_{(m,n)}}$$

with $\delta_j = 0$ if j equals 0 and $\delta_j = 1$ otherwise.

For an AES-like cipher $\mathfrak{C}$, let $\text{rnd}_{\le r}(I)$ denote the output after r rounds for the input I. A trail $\mathfrak{T}$ is said to be a *differential trail in* $\mathfrak{C}$ if there exists two $(M \times N)$ inputs I, I' to the cipher, such that $\mathfrak{T}$ represents exactely the differential state after each round. This means that $(t^{(r)}_{(m,n)})_{M\times N} = \text{rnd}_{\le r}(I) \oplus \text{rnd}_{\le r}(I')$ for each round $r > 0$ and $(t^{(0)}_{(m,n)})_{M\times N} = I \oplus I'$.

Since every round r is defined by the transformation function $\text{rnd}_{K_r,\pi_r,\texttt{MC}^{(r)}_1,\ldots,\texttt{MC}^{(r)}_N}$, as described in Definition 2.3.1, it holds that

$$\text{rnd}_{K_r,\pi_r,\texttt{MC}^{(r)}_1,\ldots,\texttt{MC}^{(r)}_N}(I) = \tilde{\text{rnd}}_{\pi_r,\texttt{MC}^{(r)}_1,\ldots,\texttt{MC}^{(r)}_N}(I) \oplus K_r.$$

for a round input I. It follows that every differential trail is independent from the used key.

Another important aspect is the definition of active S-boxes. An S-box is called *active*, if it contains a non-zero input difference. Given a differential trail, the active S-boxes are represented by its non-zero entries. Thus, the weight of a trail equals exactely the number of active S-boxes over the considered number of rounds.

The differential branch number of a transformation expresses the minimum number of active S-boxes in any two-round trail which are affected by the transformation. Thereby the trivial case with no active bundles at all is excluded. Otherwise the branch number would always be 0. Referring to [DR02], it can be defined in the following way:

Definition 3.2.2. For a linear automorphism

$$\theta : GF(2^l)^n \rightarrow GF(2^l)^n,\ (v_0, ..., v_{n-1}) \mapsto (v_0^{'}, ..., v_{n-1}^{'}),$$

the differential branch number is defined as

$$B_\theta = \min_{v \neq 0} \{ \sum_{i=0}^{n-1} (\delta_{v_i} + \delta_{v_i^{'}}) \}.$$

The bundles contain bitwise XOR-differences. Because of the linearity, the branch number can be defined with direct inputs and outputs instead of differences. This is done in the definition above.

It is obvious that for every n-dimensional linear automorphism $\theta : GF(2^l)^n \rightarrow GF(2^l)^n$, the bundle branch number B_θ is upper bounded by $n + 1$ and lower bounded by 2. Note that for an injective linear mapping, the output has at least one active bundle if the input difference is non-zero. This is due to the fact that the zero-vector is the only one that maps to 0. Hence, an active pattern cannot turn inactive. Another trivial fact is that $B_\theta = B_{\theta^{-1}}$. The branch number will be the major considered property of the `MixColumns` layer.

Definition 3.2.3. *An AES-like cipher* $\mathfrak{C}$ *has branch number* B_θ if and only if every linear mixing in $\mathfrak{C}$ has branch number B_θ.

3.2.1 Optimal Branch Numbers from MDS Codes

The linear mixing transformations can be defined using linear codes. Especially, optimal diffusion properties can be realized using maximum distance separable codes (MDS codes). A detailed introduction in coding theory can be found in [MS78]. Details on how to use such MDS codes in order to obtain optimal diffusion are explained in [RDP+96]. In the following, the most important definitions are given. They are based on the literature.

Definition 3.2.4. A *linear* $[n, k, d]_l$*-code* is a k-dimensional subspace V of the vector space $GF(2^l)^n$ such that d is the lowest number, for which every $v \in V$, $v \neq 0$ has at least d non-zero components.

A linear $[n, k, d]_l$-code is called *maximum distance separable (MDS)*, if d fulfills the singleton bound, which means that $d = n - k + 1$.

In order to cope with the mixings, the invertible square matrix $A \in GF(2^l)^{n \times n}$ *has branch number* B_θ, if the set

$$\{(v, Av) \mid v \in GF(2^l)^n\}$$

is a linear $[2n, n, B_\theta]_l$-code. For an optimal branch number of $n+1$, the matrix is called *MDS*.

The last definition is equivalent to the condition, that $G = [I \mid A]$ is a generator matrix of the corresponding linear code. Thereby, I denotes the $n \times n$ unit matrix. If and only if the invertible matrix $A \in GF(2^l)^{n \times n}$ has branch number B_θ, then the linear automorphism $v \mapsto Av$ defines a mixing transformation `MC` with branch number B_θ. In order to cope with all of the possible `MixColumns` and `SubBytes` steps within an AES-like cipher, one can make use of the following:

Lemma 3.2.5. *Let* $\texttt{MC} : GF(2^l)^n \to GF(2^l)^n$ *be an invertible linear mixing with branch number* B_θ. *Let* $v = (v_1, \ldots, v_n) \in GF(2^l)^n \setminus \{0\}$ *such that* $\texttt{MC}(v) = w = (w_1, \ldots, w_n)$. *Then for all* $a_1, \ldots, a_{2n} \in GF(2^l) \setminus \{0\}$, *one can construct an invertible linear mixing* $\texttt{MC}'$ *with branch number* B_θ *such that* $\texttt{MC}'(a_1 v_1, \ldots, a_n v_n) = (a_{n+1} w_1, \ldots, a_{2n} w_n)$.

Proof. Let $G = [I \mid A]$ be the generator matrix in standard form of the linear $[2n, n, B_\theta]_l$-code C corresponding to `MC`. Now one can construct an equivalent code C' with the same minimal distance by just multiplying every column of G by non-zero scalars $a_1, \ldots, a_{2n}$ [Wel88, pp. 54-55]. In order to obtain a generator matrix $G' = [I \mid A']$ of C' in standard form, one just scales the rows by the non-zero values $a_1^{-1}, \ldots, a_n^{-1}$. This does not change the generated code and defines the new mixing $\texttt{MC}' : x \to A'x$.

$$\begin{array}{c} \\ a_1^{-1} \\ \vdots \\ a_n^{-1} \end{array} \begin{array}{c} \begin{array}{cccccc} a_1 & \ldots & a_n & a_{n+1} & \ldots & a_{2n} \end{array} \\ \begin{pmatrix} 1 & & & & & \\ & \ddots & & & A' & \\ & & 1 & & & \end{pmatrix} \end{array}$$

If the matrix A was invertible, then A' is invertible as well since A' is obtained from A by just scaling the rows and the columns. □

3.3 Guaranteed Active S-Boxes as a Lower Bound

In order to calculate a useful lower bound on the number of active bundles in an efficient way, the idea is to only focus on the `ShiftRows` layer. The `MixColumns` operation will be considered as a black box which has the branch number property. Unfortunately, the lower bound determined in that way can be rather untight since the lowest trail that realizes this bound may not exist in a concrete AES-like instance. However, since no explicit mixing is considered, the lower bound only depends on the permutation layer and the `MixColumn` branch number. A formal definition of that restriction is given in the following.

Definition 3.3.1. The sequence of permutations $(\pi_1, \dots, \pi_R)$ with $\pi_i \in S_{M\times N}$ *tightly guarantees n active S-boxes for branch number B_θ*, if and only if n is maximal such that for all R-round $M \times N$-dimensional AES-like ciphers $\mathfrak{C}$ with branch number B_θ that use $\pi_1, \dots, \pi_R$ and for all non-trivial differential R-round trails $\mathfrak{T}$ in $\mathfrak{C}$, the weight of $\mathfrak{T}$ is lower bounded by n. For short, n is maximal such that

$$\forall \text{ AES-like ciphers } \mathfrak{C} \text{ with branch number } B_\theta \text{ using } \pi_1, \dots, \pi_R$$
$$\forall \text{ non-trivial differential trails } \mathfrak{T} \text{ in } \mathfrak{C}: \quad \text{weight}(\mathfrak{T}) \geq n.$$

This is denoted by $(\pi_1, \dots, \pi_R) \stackrel{B_\theta}{\to} n$.

This definition only describes the idea of considering the mixing layer as a black box. It is obvious that the number of guaranteed active S-boxes is always a lower bound for the actual minimum number of active bundles in any concrete AES-like cipher. For an optimal branch number, a tight bound on the amount of guaranteed active S-boxes can be computed quite easy using a mixed-integer linear programming model. It is done in the next chapter.

One has to make sure that this definition is independent of the concrete S-box functions within the AES-like ciphers. This is the case since for a given differential trail $\mathfrak{T}$ in a cipher $\mathfrak{C}$ using the bijection $\mathtt{S}$ as the S-box, one can construct a trail $\mathfrak{T}'$ in a cipher $\mathfrak{C}'$ using another S-box $\mathtt{S}'$ such that $\text{weight}(\mathfrak{T}) = \text{weight}(\mathfrak{T}')$. This construction can be realized by changing the mixing layer as stated in Lemma 3.2.5. The concrete S-box function only changes the concrete non-zero values within the trail.

Since one is interested in finding permutations that guarantee a high trail weight, it is useful to characterize them and show under which conditions the permutations are equivalent under the number of guaranteed active S-boxes. This approach reduces the problem domain, if an optimal permutation should be found. For that, a few more definitions on these permutation sequences are made.

Definition 3.3.2. Two sequences of permutations $(\pi_1, \dots, \pi_R), (\pi_1', \dots, \pi_R')$ are said to be *equivalent*, if and only if they tightly guarantee the same number of active S-boxes under the same branch number. Thus,

$$\forall B_\theta \in \{2, \dots, M+1\}: \text{If } (\pi_1, \dots, \pi_R) \stackrel{B_\theta}{\to} n \text{ and } (\pi_1', \dots, \pi_R') \stackrel{B_\theta}{\to} n', \text{ then } n = n'.$$

This is equivalence relation is expressed with the symbol $\sim$.

The sequence is called *k-alternating*, if it consists of the same k permutations in an alternating way. It is written as $(\pi_1, \dots, \pi_k)_R$.

In this work, the focus often lies on two-alternating permutations over eight rounds. Using the notation above, sequences of the form $(\pi_1, \pi_2)_8 = (\pi_1, \pi_2, \pi_1, \pi_2, \pi_1, \pi_2, \pi_1, \pi_2)$ are important.

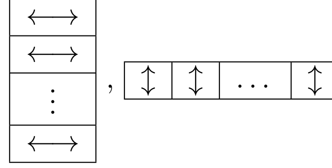

Figure 3.1: Pictograms for permutations on seperate rows or columns.

3.4 Characterizing Permutation Sequences

In order to characterize the permutations, it it obvious that for all permutations $\pi_1, \dots, \pi_R, \pi_1^{'}, \pi_R^{'} \in S_{M\times N}$ the equivalence

$$(\pi_1, \dots, \pi_R) \sim (\pi_1^{'}, \pi_2, \dots, \pi_{R-1}, \pi_R^{'})$$

holds. This is true since the number of active S-boxes over R rounds is invariant under the last permutation π_R and by reordering the initial state, the first permutation π_1 can be changed as well. Another observation is the fact that

$$(\pi_1, \dots, \pi_R) \sim (\pi_R^{-1}, \pi_{R-1}^{-1}, \dots, \pi_1^{-1}).$$

This result can be obtained by considering the cipher in decryption mode. If $\mathfrak{T}$ is a differential trail that realizes the permutations $\pi_1, \dots, \pi_R$, then a trail $\mathfrak{T}^{'}$ with the same weight is realized by the inverse permutations $\pi_R^{-1}, \dots, \pi_1^{-1}$ for the cipher with the inverse `MixColumns` layer. These trails only differ by a permutation and a substitution of the bundle entries.[1].

Theorems 3.4.2 and 3.4.5 state some more of these equivalences, which will be quite helpful in the later analysis. Especially Theorem 3.4.5 provides a kind of standard form for (k-alternating) permutation sequences. At first, we define the following.

Definition 3.4.1. A permutation $\pi \in S_{M\times N}$ *operates on the rows seperately*, if and only if

$$\pi(m, n) = (m, \pi^{\beta}(m, n))$$

for all rows m and all columns n.

A permutation $\pi \in S_{M\times N}$ *operates on the columns seperately*, if and only if

$$\pi(m, n) = (\pi^{\alpha}(m, n), n)$$

for all rows m and all columns n.

Figure 3.1 illustrates how such permutations operate. It would be interesting to know, if one is able construct equivalences of permutation layers due to these special permutations on the rows or columns. Fortunately, there are some useful properties which will be presented in the following theorems. The first one concentrates on the columns.

[1] Of course, the inverse of the S-box has to be used as well. However, we have proven that the guaranteed trail weight is independent of the used S-box.

Theorem 3.4.2. *The number of tightly guaranteed active S-boxes is invariant under permutations that operate on the columns seperately.*

That means for all $\pi, \pi^{'} \in S_{M\times N}$ *operating on the columns seperately and all* $r \in \{1,\dots,R\}$ *the equivalence*

$$(\pi_1,\dots,\pi_R) \sim (\pi_1,\dots,\pi^{'} \circ \pi_r \circ \pi,\dots,\pi_R)$$

holds. Especially, whole rows can be arbitrarily permuted.

Proof. Fix a round r and a branch number B_θ and let the sequence $(\pi_1,\dots,\pi_R)$ tightly guarantee n active S-boxes. Let $\pi^{'}$ be a permutation that operates on the columns seperately. At first we show, that $(\pi_1,\dots,\pi^{'} \circ \pi_r,\dots,\pi_R) \overset{B_\theta}{\to} n$ as well.

Let $\mathfrak{C}$ be an arbitrary R-round AES-like cipher which uses the round transformations

$$\mathrm{rnd}^{(j)}_{K_j,\pi_j} = \texttt{AddRoundKey}_{K_j} \circ (\texttt{MC}^{(j)}_1,\dots,\texttt{MC}^{(j)}_N) \circ \texttt{Permute}_{\pi_j} \circ \texttt{SubBytes}$$

for round $j \neq r$ and

$$\mathrm{rnd}^{(r)}_{K_r,\pi^{'}\circ\pi_r} = \texttt{AddRoundKey}_{K_r} \circ (\texttt{MC}^{(r)}_1,\dots,\texttt{MC}^{(r)}_N) \circ \texttt{Permute}_{\pi^{'}\circ\pi_r} \circ \texttt{SubBytes}$$

for round r. It is to show that the weight of any differential R-round trail $\mathfrak{T}$ in $\mathfrak{C}$ is lower bounded by n. It holds that

$$\mathrm{rnd}^{(r)}_{K_r,\pi^{'}\circ\pi_r} = \texttt{AddRoundKey}_{K_r} \circ [(\texttt{MC}^{(r)}_1,\dots,\texttt{MC}^{(r)}_N) \circ \texttt{Permute}_{\pi^{'}}] \circ \texttt{Permute}_{\pi_r} \circ \texttt{SubBytes}\,.$$

Since $\pi^{'}$ operates on the columns seperately,

$$(\texttt{MC}^{(r)'}_1,\dots,\texttt{MC}^{(r)'}_N) := (\texttt{MC}^{(r)}_1,\dots,\texttt{MC}^{(r)}_N) \circ \texttt{Permute}_{\pi^{'}}$$

defines a linear `MixColumns` step. Every $\texttt{MC}^{(r)'}_j$ is just a permutation of $\texttt{MC}^{(r)}_j$ and thus also linear and results in the same output weight (and therefore has the same branch number). Using this round transformation, $\mathfrak{C}$ becomes a cipher which uses the permutations $\pi_1,\dots,\pi_R$. Since $(\pi_1,\dots,\pi_R) \overset{B_\theta}{\to} n$, any differential R-round trail in $\mathfrak{C}$ is lower bounded by n and n is tight.

With the same argument, it can be proven that the sequence $(\pi_1,\dots,\pi_r \circ \pi^{'},\dots,\pi_R) \overset{B_\theta}{\to} n$ as well. One can assume that $r \geq 2$. For $r = 1$ there is nothing to show, since the number of guaranteed active S-boxes is independent from the first permutation. If $r \geq 2$, the $\texttt{Permute}_{\pi^{'}}$ step commutes to the previous round and

$$(\texttt{MC}^{(r-1)'}_1,\dots,\texttt{MC}^{(r-1)'}_N) := \texttt{Permute}_{\pi^{'}} \circ (\texttt{MC}^{(r-1)}_1,\dots,\texttt{MC}^{(r-1)}_N)$$

defines the new general `MixColumns` step. □

In order to prove a very useful theorem which provides a kind of standard form on the round permutations, one can make use of the following two results. The first one is a combinatorial lemma on permutations.

Lemma 3.4.3. *Every permutation $\pi \in S_{M\times N}$ can be represented as $\pi^{'}_{col} \circ \pi_{row} \circ \pi_{col}$ where $\pi_{col}, \pi^{'}_{col}$ are operating on columns seperately and π_{row} is operating on rows seperately. As an illustration*

$$\pi = \boxed{\updownarrow}\boxed{\updownarrow}\boxed{\ldots}\boxed{\updownarrow} \circ \begin{array}{|c|}\hline \longleftrightarrow \\ \hline \longleftrightarrow \\ \hline \end{array} \circ \boxed{\updownarrow}\boxed{\updownarrow}\boxed{\ldots}\boxed{\updownarrow}.$$

This lemma can be proven in a constructive way if the number of rows is a power of 2.

Proof. (for $M = 2^d$) This is done by induction over d.

Let $d = 1$: Define the N multisets

$$T_i := \{\pi^\beta(x, i) \mid x \in \{0, 1\}\} = \{\pi^\beta(0, i), \pi^\beta(1, i)\}$$

for $i = 0, \ldots, N-1$. These multisets are representing the target column of the two elements in column i. Every column occures twice over all T_i since π is a permutation. Thus $\biguplus_{i=0}^{N-1} T_i = \{0, 0, 1, 1, \ldots, N-1, N-1\}$. The major goal is to find a selection vector $c = (c_0, \ldots, c_{N-1})$ with $c_i \in \{0, 1\}$ and select $R^0, R^1 \in T_0 \times T_1 \times \cdots \times T_{N-1}$ as

$$R^0 = (\pi^\beta(c_0, 0), \pi^\beta(c_1, 1), \ldots, \pi^\beta(c_{N-1}, N-1))$$
$$R^1 = (\pi^\beta(\bar{c}_0, 0), \pi^\beta(\bar{c}_1, 1), \ldots, \pi^\beta(\bar{c}_{N-1}, N-1)),$$

such that R^0 contains all different components $n \in \{0, \ldots, N-1\}$. (Then it follows that R^1 fulfills the same property.) From this separation, it becomes obvious how to build the first permutation on the columns (permute in such a way that the selections are in separate rows). Since the other two permutations would be straightforward, the proof would be done. In order to construct the selections R^0 and R^1, one can assume without loss of generality that there is no T_i of the form $\{b, b\}$ for a certain b. Otherwise, c_i can be set to b and because b occured already twice, the problem could be reduced to a smaller column dimension. With the same argument, one can assume without loss of generality that for every subset $K \subsetneq \{0, \ldots, N-1\}$, the union $\biguplus_{i\in K} T_i \neq \{b_1, b_1, b_2, b_2, \ldots, b_l, b_l\}$. Otherwise, the complement $\biguplus_{i\notin K} T_i$ would be of a similar form and the problem could again be reduced into two distinct smaller ones. With these assumptions one can construct the selection as follows:

For the N multisets $T_0, \ldots, T_{N-1}$, find an index permutation $\tau \in S_N$ and arrange the T_i such that $T_{\tau(i)} = \{b_i, b_{i+1}\}$ for all $i = 0, \ldots, N-2$ and $T_{\tau(N-1)} = \{b_{N-1}, b_0\}$. The possibility of this arrangement is due to the assumptions made above. It holds that every $b_i \neq b_j$ for all $j < i$. If this construction was not possible, there would exist such a subset K as described above since every element only occures twice.

Now construct R^0 by setting $R^0_{\tau(i)} = b_i$.

	$\tau(0)$	$\tau(1)$	$\tau(2)$	$\ldots$	$\tau(N-2)$	$\tau(N-1)$
R^0	b_0	b_1	b_2	$\ldots$	b_{N-2}	b_{N-1}
R^1	b_1	b_2	b_3	$\ldots$	b_{N-1}	b_0

In order to show the induction step $d \to d+1$, a state dimension of the form $2^{d+1} \times N$ is seperated into distinct instances of dimension $2^d \times N$ as follows:

Rearrange the index state for that dimension with 2^{d+1} rows in the form

$$\begin{matrix} (0,0) & (2,0) & \dots & (2^{d+1}-2,0) & (0,1) & (2,1) & \dots & \dots & (2^{d+1}-2,\text{N-1}) \\ (1,0) & (3,0) & \dots & (2^{d+1}-1,0) & (1,1) & (3,1) & \dots & \dots & (2^{d+1}-1,\text{N-1}) \end{matrix}$$

obtaining a $2 \times 2^d N$ instance. This can be solved since the number of rows equals 2. Note that the resulting middle permutation is not operating on the rows seperately since (0,0) lies in a different row than (2,0). Instead, it operates on two distinct $2^d \times N$ instances seperately. This problem can be solved by the assumption of the induction. □

Unfortunately, the same argument does not work that easy on arbitrary row dimensions since every element occurs more than twice. In addition, the notation would be even worse. However, within personal communication, J. P. Steinberger provided a very elegant method to prove this lemma for arbitrary row dimensions. His idea was the following:

The N target multisets are defined as above for arbitrary M, in particular

$$T_i := \{\pi^\beta(x,i) \mid x \in \{0,\dots,M-1\}\} = \{\pi^\beta(0,i), \pi^\beta(1,i), \dots, \pi^\beta(M-1,i)\}.$$

Now one can define the $N \times N$ matrix $\mathcal{O}$ such that $\mathcal{O}_{i,j}$ is equal to the number of occurences of j in T_i. In order to show the existence of the three permutations, the idea is to find a row permutation step π_{row} which corresponds to this matrix. Because of the permutation property, $\mathcal{O}$ is an integer magic square matrix, which means that each row and each column sums up to M. By the Birkhoff-von Neumann Theorem [ADH98, 164], the matrix $\mathcal{O}$ decomposes into M permutation matrices $P_1, \dots, P_M$. Thus, it can be written as

$$\mathcal{O} = P_1 + P_2 + \cdots + P_M.$$

These permutation matrices correspond to the M row permutations which have to be executed in the π_{row} step.

The next result illustrates how the guaranteed trail weight is invariant under column swaps.

Lemma 3.4.4. *The number of guaranteed active S-boxes is invariant under a permutation of columns in the following way:*

Given the R permutations $\pi_1, \dots, \pi_R \in S_{M\times N}$. For all $r \in \{1,\dots,R-1\}$ and for all $i,j \in \{1,\dots,N\}$, the equivalence

$$(\pi_1,\dots,\pi_R) \sim (\pi_1,\dots,\text{swap}_{ij} \circ \pi_r, \pi_{r+1} \circ \text{swap}_{ij}, \dots, \pi_R)$$

holds. Thereby the permutation swap_{ij} swaps the whole columns i and j.

Especially, for a k-alternating sequence of permutations one obtains

$$(\pi_1,\dots,\pi_k)_R \sim (\text{swap}_{i_1 j_1} \circ \pi_1, \text{swap}_{i_2 j_2} \circ \pi_2 \circ \text{swap}_{i_1 j_1}, \dots, \pi_k \circ \text{swap}_{i_{k-1} j_{k-1}}).$$

Proof. Let $r \leq R-1$ and let swap_{ij} be an arbitrary swap of columns within the state. Without loss of generality, let $i < j$. Since $\text{swap}_{ij} \circ \text{swap}_{ij} = id$ it follows that

$$\text{swap}_{ij} \circ \text{swap}_{ij} \circ \pi_r = \pi_r$$

and thus, the equivalence

$$(\pi_1, \dots, \pi_R) \sim (\pi_1, \dots, \text{swap}_{ij} \circ \text{swap}_{ij} \circ \pi_r, \pi_{r+1}, \dots, \pi_R)$$

holds. Round r in a corresponding AES-like cipher $\mathfrak{C}$ can be written in the form

$$\texttt{AddRoundKey}_{K_r} \circ (\texttt{MC}_1^{(r)}, \dots, \texttt{MC}_N^{(r)}) \circ \texttt{Permute}_{\text{swap}_{ij}} \circ \texttt{Permute}_{\text{swap}_{ij} \circ \pi_r} \circ \texttt{SubBytes}\,.$$

Changing the round key and the mixing layer transformation, the same round transformation can be written as

$$\texttt{Permute}_{\text{swap}_{ij}} \circ \texttt{AddRoundKey}_{K_r'} \circ (\texttt{MC}_1^{(r)'}, \dots, \texttt{MC}_N^{(r)'}) \circ \texttt{Permute}_{\text{swap}_{ij} \circ \pi_r} \circ \texttt{SubBytes}\,.$$

The permutation $\texttt{Permute}_{\text{swap}_{ij}}$ almost commutes with the mixing layer $(\texttt{MC}_1^{(r)}, \dots, \texttt{MC}_N^{(r)})$ and the key addition $\texttt{AddRoundKey}_{K_r}$. The only difference is that the columns i and j are swapped. Thus, the new mixing is of the form

$$(\texttt{MC}_1^{(r)'}, \dots, \texttt{MC}_N^{(r)'}) = (\texttt{MC}_1^{(r)}, \dots, \texttt{MC}_j^{(r)}, \dots, \texttt{MC}_i^{(r)}, \dots, \texttt{MC}_N^{(r)})$$

and for the round key, it is $K_r' = \text{swap}_{ij}(K_r)$.

Commuting $\texttt{Permute}_{\text{swap}_{ij}}$ with $\texttt{SubBytes}$ in the next round, $\mathfrak{C}$ becomes a cipher which uses the permutations

$$\pi_1, \dots, \text{swap}_{ij} \circ \pi_r, \pi_{r+1} \circ \text{swap}_{ij}, \dots, \pi_R$$

and thus the equivalence follows. □

3.4.1 A Standard Form for Describing Permutation Layers

With these results and the use of Theorem 3.4.2, a very useful theorem can be proven. It can be used in order to reduce the problem domain of a brute force approach on finding optimal (k-alternating) permutation sequences and can be considered as a simple standard form for all possible permutations.

Theorem 3.4.5. *Let the state be in dimension $M \times N$ and suppose that $(\pi_1, \dots, \pi_k)_R$ is a k-alternating sequence of permutations. It is possible to construct permutations $\tilde{\pi}_1, \dots, \tilde{\pi}_k$ with*

(i) $\tilde{\pi}_1, \dots, \tilde{\pi}_k$ operate on the rows seperately,

(ii) $\tilde{\pi}_1, \dots, \tilde{\pi}_{k-1}$ leave the first row unpermuted,

such that $(\pi_1, \dots, \pi_k)_R \sim (\tilde{\pi}_1, \dots, \tilde{\pi}_k)_R$.

Proof. Because of Lemma 3.4.3, each permutation π_i can be represented as $\pi_i^{''} \circ \tilde{\pi}_i \circ \pi_i^{'}$ such that $\pi_i^{'}, \pi_i^{''}$ are operating on the columns seperately and $\tilde{\pi}_i$ operates on the rows seperately. By using Theorem 3.4.2, the equivalence $(\pi_1, \ldots, \pi_k)_R \sim (\tilde{\pi}_1, \ldots, \tilde{\pi}_k)_R$ holds and *(i)* is fulfilled. Property *(ii)* can be obtained by applying Lemma 3.4.4 until the first row is unpermuted. Since all the column swaps are permutations which operate on the rows seperately, this will not affect property *(i)*. □

Without loss of generality, one can assume that every considered permutation sequence is in that form. Every permutation can now be described by its permutations on the M rows. Furthermore, these M row permutations in one π_r can arbitrarily be swapped since $[\text{swapRows}_{ij} \circ \pi_r \circ \text{swapRows}_{ij}]$ results in an equivalent sequence after Theorem 3.4.2. These results also help later in the experimental part when cyclic rotatation within the rows are analyzed.

If the consideration is not restricted to k-alternating structures, one can assume the first row always as unpermuted (for all π_i) since the last permutation does not affect the guaranteed trail weight.

4 Computing Guaranteed Trail Weights

Since the minimum amount of active S-boxes within a cipher is a good indicator for the resistance against differential (and linear) attacks, the main goal is to guarantee high lower bounds on the number of active bundles for a fixed number of rounds. Therefore, the permutation layer should be chosen in an optimal way. In order to calculate this lower bound for a chosen permutation layer, one can use a technique called mixed-integer linear programming (MILP). In this chapter, the method is presented and applied on different shiftings and permutations in AES-like ciphers.

The Four-Round Propagation Theorem on AES [DR02, p. 140] states that the weight of every non-zero four-round trail is at least B_θ^2, if the permutation layer has optimal diffusion properties. Since the focus lies on states of the form $M \times M^2$, more rounds have to be considered. By iterating the theorem using an iterated superbox approach, one can expect an optimum of B_θ^3 active S-boxes over eight rounds for cubed states. [BK14] already provided a permutation layer which guarantees this bound. Thus, mainly eight-round patterns are considered within this work.

For this experimental work, one can use the results of the previous chapter, especially Theorem 3.4.5. Only permutation sequences in this standard form need to be considered. If solutions are provided, equivalent possibilites due to this standard form may be left out in the presentation. Since one is interested in using permutation layers with good implementation advantages, cyclic rotations within the rows (analogous to the `ShiftRows` layer) play an important role in the experiments.

4.1 Mixed-Integer Linear Programming

Mixed-integer linear programming is a specialization of a linear optimization problem with the goal to minimize or maximize a linear objective function due to some linear constraints. Linear programming is used in many different areas, like economics, logistics, or communication [Kal02].

The problems considered in this thesis are quite easy to describe. For that reason, a very simple definition of a mixed-integer linear programming model will be given.

Definition 4.1.1. A *MILP model* consists of a finite set of n variables v_i where each one can be defined ether as a non-negative integer or a non-negtive rational, an objective function $\sum_{i<n} a_i v_i$ with $a_i \in \mathbb{Q}$ and a finite set of constraints $\{\sum_{i<n} a_i v_i \leq a_n\}$ with $a_i \in \mathbb{Q}$. The *solution of the MILP model* (if it exists) is the minimum of the objetive function such that all constraints hold.

Note that one can easily construct maximization problems by inverting the coefficients. In the problems presented here, only variables of integer type are used.

The main idea is to add variables for each bundle over the considered number of rounds and to describe the cipher by the constraints in such a way that the variables will be 0 if and only if the bundles are inactive. The objective function is then defined by the sum of all variables representing the bundles. This approach is shown in [MWGP11] and is presented in the following. The method is also suitable for other block ciphers and not restricted to AES structures.

4.2 The Cipher as a Mixed-Integer Linear Programming Model

Given an AES-like state dimension $M \times N$, permutation layers π_r depending on the round and the `MixColumns` branch number $B_\theta \in \{2, \dots, M+1\}$, a MILP model can be constructed which has as the solution a lower bound on the number of active S-boxes for a considered number of rounds. The number of considered rounds is denoted by R.

The first step is to add $(R+1) \cdot M \cdot N$ non-negative integer variables $c^{(r)}_{m,n}$ for each state bundle at the beginning of each considered round. They will represent the activity of the S-box. The output of the last considered round will be represented as well, thus $c^{(R)}_{m,n}$ is added. The $m \in \{0, ..., M-1\}$ and the $n \in \{0, ..., N-1\}$ denote the index of the bundle within the state and r is the round index. Before the encryption starts, r equals 0 and therefore the $c^{(0)}_{m,n}$ represent whether the input difference is 0 or non-zero for the particular state bundle.

The objective function should correspond to the sum of all active S-boxes over all considered rounds and is therefore defined as

$$\sum_{\substack{0 \leq m < M \\ 0 \leq n < N \\ 0 \leq r < R}} c^{(r)}_{m,n}.$$

The constraints have to make sure that the variables will equal 0 if and only if the corresponding S-box is inactive and that they will equal 1 otherwise. Since the key additions do not have any effect on the difference pattern and thus also not on the activity, they do not have to be considered in the constraints. The same holds for the `SubBytes` layer. The only relevant components are permutation and mixing. The cipher can be considered step by step. In the beginning, one starts with the first column in the first round. The bundles are permuted by π_0 such that the variables $(c^{(0)}_{\pi_0(0,0)}, ..., c^{(0)}_{\pi_0(M-1,0)})$ represent the column after that permutation. The `MixColumns` branch number guarantees a sum of B_θ active bundles in the column for two consecutive rounds, supposing that at least one is active at all. The naive way to describe that constraint would be

$$c^{(0)}_{\pi_0(0,0)} + ... + c^{(0)}_{\pi_0(M-1,0)} + c^{(1)}_{0,0} + ... + c^{(1)}_{M-1,0} \geq B_\theta.$$

This description is not sufficient since the possibility of no active bundles at all in that column is not considered. To cope with that issue, a new integer variable $d^{(r)}_n$ for each column in each round has to be added. It is called dummy variable. In the example

of the first column, $d_0^{(0)}$ is added. The dummy variable should be 0 if and only if the column has no active bundles and 1 otherwise. Thus, the following constraints are added:

$$\begin{aligned}
c^{(0)}_{\pi_0(0,0)} + ... + c^{(0)}_{\pi_0(M-1,0)} + c^{(1)}_{0,0} + ... + c^{(1)}_{M-1,0} &\geq d_0^{(0)} \cdot B_\theta \\
c^{(0)}_{\pi_0(0,0)} &\leq d_0^{(0)} \\
&\vdots \\
c^{(0)}_{\pi_0(M-1,0)} &\leq d_0^{(0)} \\
c^{(1)}_{0,0} &\leq d_0^{(0)} \\
&\vdots \\
c^{(1)}_{M-1,0} &\leq d_0^{(0)}.
\end{aligned}$$

If one wants to allow non-optimal branch numbers ($B_\theta < M + 1$), one will have to make sure that an active column cannot be turned inactive or vice versa. This could be the case if the input- or output column already has B_θ bundles active. It has to be prevented by adding the two constraints

$$\begin{aligned}
d_0^{(0)} &\leq c^{(0)}_{\pi_0(0,0)} + ... + c^{(0)}_{\pi_0(M-1,0)} \\
d_0^{(0)} &\leq c^{(1)}_{0,0} + ... + c^{(1)}_{M-1,0}
\end{aligned}$$

for the example of the first column.

This exessive adding of new variables and constraints is done for all of the columns in every round, resulting in a rather long system of inequations. The properties of the constraints guarantee a maximum value of 1 for each variable since the objective function shall be minimized. However, if one wanted to solve this model, the solution would always equal 0 since the case of no active S-boxes at all minimizes the objective function. Because of that, one final constraint is added, making sure that at least one bundle is active. This is

$$\sum_{\substack{0 \leq m < M \\ 0 \leq n < N \\ 0 \leq r < R}} c^{(r)}_{m,n} \geq 1.$$

To sum up all the steps described above, the following algorithm shows how the MILP model is created. It is based on the paper by Mouha et. al. However, the version presented here is more generalized in order to allow arbitrary permutations, state dimensions and branch numbers.

Algorithm 4.2.1: MILPgen$_{(\pi_1,\dots,\pi_R),B_\theta,M,N}$

input : permutations $\pi_1,\dots,\pi_R$, `MixColumns` branch number B_θ, dimensions M, N
output : mixed-integer linear programming model MODEL$_{(\pi_1,\dots,\pi_R),B_\theta,M,N}$

1 **forall the** *rounds* $r \in \{1,\dots,R+1\}$ **do**
2   state$_r$ $\longleftarrow$ int$[M][N]$;
3   **forall the** *rows* $m < M$*, columns* $n < N$ **do**
4   state$_r[m][n] = m + Nn + (r-1)MN$;
5 model $\longleftarrow$ initializeMILP;
6 **for** *i=0* **to** $(R+1)MN-1$ **do** model.addVariable(c_i);
7 **for** *i=0* **to** $RN-1$ **do** model.addVariable(d_i);
8 model.setObjective($\sum_{i=0}^{RMN-1} c_i$);
9 **forall the** *rounds* $r \in \{1,\dots,R\}$ **do**
10   permuteState(state$_r, \pi_r$);
11   **forall the** *columns* $n < N$ **do**
12   model.addConstraint($\sum_{m=0}^{M-1} c_{\text{state}_r[m][n]} + \sum_{m=0}^{M-1} c_{\text{state}_{r+1}[m][n]} \geq B_\theta d_{(r-1)N+n}$);
        forall the *rows* $m < M$ **do**
13   model.addConstraint($d_{(r-1)N+n} - c_{\text{state}_r[m][n]} \geq 0$);
14   model.addConstraint($d_{(r-1)N+n} - c_{\text{state}_{r+1}[m][n]} \geq 0$);
15   **if** $B_\theta < M+1$ **then**
16   model.addConstraint($\sum_{m=0}^{M-1} c_{\text{state}_r[m][n]} - B_\theta d_{(r-1)N+n} \geq 0$);
17   model.addConstraint($\sum_{m=0}^{M-1} c_{\text{state}_{r+1}[m][n]} - B_\theta d_{(r-1)N+n} \geq 0$);
18 model.addConstraint($\sum_{i=0}^{RMN-1} c_i \geq 1$);
19 **return** model;

It would be a very nice result, if this approach exactely computed the number of tightly guaranteed active bundles. In fact, this is proven in the following for the case of optimal branch numbers. In order to show that the MILP model outputs such a tight bound, one can make use of the following lemma.

Lemma 4.2.1. *Let* $\log_2(n+2) < l$ *and let* C *be a linear* $[2n,n]_l$*-code which is MDS. For every subset* $S \subseteq \{1,\dots,2n\}$ *with* $n+1 \leq |S| \leq 2n$*, there exists a vector* $v = (v_1,\dots,v_{2n}) \in C$ *such that* $v_i \neq 0$ *if and only if* $i \in S$*.*

Proof. Define two subsets $S_1, S_2 \subseteq S$ such that $|S_1| = |S_2| = n+1$ and $S_1 \cup S_2 = S$. This is possible since $|S| \geq n+1$. From Theorem 4 in [MS78, p. 319] it follows that there exist two vectors $v^{(1)} = (v_1^{(1)},\dots,v_{2n}^{(1)})$ and $v^{(2)} = (v_1^{(2)},\dots,v_{2n}^{(2)})$ in C such that $v_i^{(j)} \neq 0$ if and only if $i \in S_j$. Now, one can construct v as a linear combination $v := v^{(1)} + cv^{(2)}$ with $c \in GF(2^l)$ as follows:

Choose $c \neq 0$ such that for all non-zero components $v_i^{(1)}$ in $v^{(1)}$ the identity

$$c \cdot v_i^{(2)} \neq -v_i^{(1)}$$

holds. This is possible because of the field property of $GF(2^l)$ and since $2^l > n + 2$. □

Given a concrete MDS transformation (which has a sufficiently large dimension), every Boolean activity pattern which fulfills the branch number property can be be realized. Boolean means that the value is either 0 or non-zero depending on the activity. Hence, the concrete entries within the trail are not considered. However, for all possible concrete values, a corresponding MDS transformation can be constructed as Lemma 3.2.5 already states. One obtains as a corollary:

Corollary 4.2.2. *Let* $\log_2(n + 2) < l$ *and let an MDS matrix* $A \in GF(2^l)^{n\times n}$ *exist. Then for all* $v, w \in GF(2^l)^n$ *with* $\mathrm{weight}(v) + \mathrm{weight}(w) \geq n + 1$*, there exists an MDS matrix* $\tilde{A} \in GF(2^l)^{n\times n}$ *such that* $w = \tilde{A}v$*.*

With these results one can prove the following.

Theorem 4.2.3. *The solution of* $\mathrm{MODEL}_{(\pi_1,\dots,\pi_R),B_\theta}$ *is always a lower bound on the number of guaranteed active S-boxes for the branch number* B_θ *of the sequence* $(\pi_1, \dots, \pi_R)$*. If the branch number is optimal for the given dimensions and supposed a linear mixing with this branch number exists (and the bit length* $l > \log_2(M + 2)$*), the solution is exately the maximum number of guaranteed active S-boxes.*

Proof. Corollary 4.2.2 shows that every pattern, which fulfills the branch number property during the mixing steps, is a possible trail for a certain linear MDS layer. This can be constructed, if only one mixing with an optimal branch number exists. For every round, the `MixColumns` input pattern is given by the S-box outcome and the `ShiftRows` step.

Since every possible `MixColumns` outcome (fulfilling the branch number property) can be realized, the program iterates exactely over all concrete possibilites. □

4.3 A First Approach on the Minimization Problem

As a first approach on solving such a problem for AES-like ciphers, a C program is written that creates all the constraints and the objective function of the MILP-model for given permutations, a given state dimension and `MixColumns` branch number. It is solved with the commercial Gurobi solver. The program implementing the algorithm and thus generating the MILP model is based on the simple example code in [MWGP11]. The tool is written with the use of the C interface of Gurobi. For a documentation and examples see [go]. Listing A.1 in the appendix shows how it is implemented.

Before using the commercial gurobi solver, the MILP models have been tried to solve with the MILP function in sage [sag]. It uses the open source GNU Linear Programming Kit (GLPK) solver by default. Unfortunately, it is unable to solve lots of the models in sufficient time, especiallly in the dimension 4×16. For these tests, all the GLPK default settings were used. In comparison, Gurobi (with default settings) is significantly faster and able to solve some models within seconds, for which GLPK is still computing after lots of hours. For details on the efficiency of different MILP solvers, a general comparison can be found in [MT12].

In the following, the program is tested for a few examples in the three smallest cubed state dimensions. At first, mainly cyclic rotations within the rows and cube turns are considered for the (at most) two-alternating case. The models are tried to solve with the GLPK (v 4.44) and Gurobi solver (v 5.6.2). All experiments are done on an Intel Celeron 1007U dual core CPU with 1.50GHz clock frequncy each. For all of the tests, the optimal branch number of $M+1$ is considered and the solution is computed over four- and eight-round patterns. Note that the state can be represented two-dimensional ($M \times M^2$) or in 3D as a cube ($M \times M \times M$) as described in Figure 2.2.

Considering the notation, s_1-...-s_M denotes a cyclic (left) rotation within the rows (state is represented two-dimensional) by the offsets $s_1, \ldots, s_M$. For example, the permutation 0-1-2-3 is similar to the standard `ShiftRows` step. However, we also define a `SR` permutation. If the state is represented in the 3D form as a cube, this permutation operates on all the M distinct square state instances separately doing the `ShiftRows` operation on each of these states. Figure 4.1 briefly illustrates the difference between these two permutation types.

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \end{pmatrix} \xrightarrow{\text{0-1}} \begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,1} & a_{1,2} & a_{1,3} & a_{1,0} \end{pmatrix}$$

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \end{pmatrix} \xrightarrow{\texttt{SR}} \begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,1} & a_{1,0} & a_{1,3} & a_{1,2} \end{pmatrix}$$

Figure 4.1: Illustration of the permutations 0-1 and `SR` for dimension 2×4.

Considering the whole cube, rot(120°) expresses a 120° rotation of the (3D) state by its diagonal axis and rot(90°) denotes a simple 90° turn to the right. Of course, the three-dimensional rotation is not in the standard form.

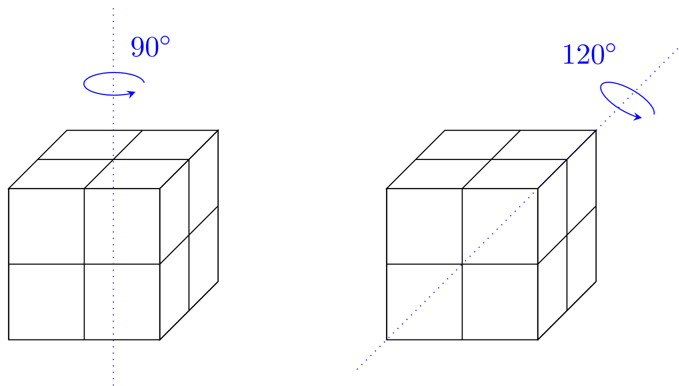

Figure 4.2: Illustration of the cube rotations rot(90°) and rot(120°).

4.3.1 Dimension 2 x 4

This is the most simple cubed AES-like state arrangement and therefore quite easy to analyze in comparison to higher dimensions. The optimal `MixColumns` branch number equals 3 in this case. The execution time is given for the eight-round case with default parameter settings. All of the models can be solved even with the GLPK solver in less than 1 second.

Table 4.1: Mimimum number of active bundles in a 2×4 state for a few examples.

Permutation π_1	**Permutation** π_2	**4-round minimum**	**8-round minimum**	**Ex. time: Gurobi**
id	id	6	12	0.12s
0-1	π_1	9	24	0.19s
0-2	π_1	9	18	0.09s
0-1	0-2	9	24	0.16s
0-1	1-0	9	24	0.2s
0-2	`SR`	9	24	0.13s
rot(120°)	0-2	9	24	0.15s
0-2	rot(120°)	9	**27**	0.21s
$\pi_2 \circ$ rot(90°)	0-2	9	24	0.2s
0-2	$\pi_1 \circ$ rot(90°)	9	**27**	0.14s
$\pi_2 \circ$ rot(90°)	`SR`	9	24	0.15s
`SR`	$\pi_1 \circ$ rot(90°)	9	**27**	0.16s

4.3.2 Dimension 3 x 9

In this arrangement, the GLPK solver (with default settings) is not sufficient anymore. No time values indicate that there was still no result after 2 hours of computing for the specific solver. Table 4.2 illustrates the results. The `MixColumns` branch number equals 4 in all of the tests.

4.3.3 Dimension 4 x 16

This is the most complex state considered and an important one for practical use. It can be considered as an extension of the standard AES dimension. The optimal `MixColumns` branch number equals 5 and the results are presented in Table 4.3. In case the solution time took longer than 30 seconds, it is rounded in 5-second steps.

One can notice some interesting facts in these experiments. The permutation layers which have optimal diffusion power cause at least B_θ^2 active S-boxes in any four-round trail. This is an experimental validation of the Four-Round Propagation Theorem for the AES. However, the minimum number of active bundles over eight rounds can be

Table 4.2: Mimimum number of active bundles in a 3 × 9 state for a few examples.

Permutation π_1	**Permutation** π_2	**4-round minimum**	**8-round minimum**	**Ex. time: GLPK**	**Ex. time: Gurobi**
id	id	8	16	<1s	0.21s
0-1-2	π_1	16	32	<5s	1.06s
0-2-4	π_1	16	32	<5s	0.92s
0-3-6	π_1	16	32	<5s	0.5s
0-1-2	0-3-6	16	48	<10s	1.02s
0-3-6	SR	16	48	<10s	0.79s
rot(120°)	0-3-6	16	48	<10s	0.76s
0-3-6	rot(120°)	16	**64**		0.85s
$\pi_2 \circ$ rot(90°)	0-3-6	16	48	<20s	0.95s
0-3-6	$\pi_1 \circ$ rot(90°)	16	**64**		0.58s
$\pi_2 \circ$ rot(90°)	SR	16	48	<10s	1.23s
SR	$\pi_1 \circ$ rot(90°)	16	**64**		1.08s

Table 4.3: Mimimum number of active bundles in a 4 × 16 state for a few examples.

Permutation π_1	**Permutation** π_2	**4-round minimum**	**8-round minimum**	**Ex. time: GLPK**	**Ex. time: Gurobi**
id	id	10	20	<5s	0.64s
0-1-2-3	π_1	25	60	∼ 26m 30s	8s
0-2-4-6	π_1	25	60	∼ 19m 25s	4.72s
0-3-6-9	π_1	25	60	∼ 23m 25s	7.69s
0-4-8-12	π_1	25	50	∼ 50s	1.78s
0-1-2-3	0-4-8-12	25	80	∼ 8m	8.12s
0-4-8-12	SR	25	80	∼ 4m 15s	5.28s
rot(120°)	0-4-8-12	25	80	∼ 8m 35s	8.64s
0-4-8-12	rot(120°)	25	**125**		12.56s
$\pi_2 \circ$ rot(90°)	0-4-8-12	25	80		8.24s
0-4-8-12	$\pi_1 \circ$ rot(90°)	25	**125**		2.57s
$\pi_2 \circ$ rot(90°)	SR	25	80	∼ 19m	8.88s
SR	$\pi_1 \circ$ rot(90°)	25	**125**		12.41s

different. When using two different permutations in an alternating way, the order of them is relevant for the solution of the MILP model.

The last six experiments in each table show three equivalent optimal permutation layers and their reordering. The equivalence follows from the equivalence of the standard form described in Theorem 3.4.5 and from just changing the rotation axis of the cyclic rotations in the last case. For dimension 4 × 16, it is similar to the AESQ, the result described in [BK14] and leads to an optimal lower bound of 125 active bundles over eight rounds. The only difference is the column shuffle-permutation instead of the cube

rotation, which is executed every second round. Just like the cube turn, the shuffle distributes every column from one quadratic state instance over all four states. This is exactely the same superbox argument which proves the bound of B_θ^3.

4.4 Finding Optimal Permutations

One question is what the optimal reachable bound for the guaranteed trail weight is. In particular, whether the assumed optimal bound of B_θ^3 over eight rounds is really the optimum. It is quite easy to see that the four-round optimum is B_θ^2. That means that there is no state dimension and no permutation layer which provides a tightly guaranteed four-round trail weight of more than B_θ^2.

Supposed a four-round trail $\mathfrak{T} = (t^{(0)}, t^{(1)}, t^{(2)}, t^{(3)}, t^{(4)})$ is given such that

$$\text{weight}(t^{(1)}) = 1 \text{ and } \text{weight}(t^{(0)}) = \text{weight}(t^{(2)}) = B_\theta - 1.$$

For any state dimension and any permutation layer, there exists a cipher $\mathfrak{C}$ such that this is a trail in $\mathfrak{C}$. Since the permutation activates at most $B_\theta - 1$ colums in $t^{(3)}$ and every active column only needs to contain at most $B_\theta - 1$ active bundles, one can assume that $\text{weight}(t^{(3)}) \leq (B_\theta - 1)^2$ as well. To sum it up, it is

$$\text{weight}(\mathfrak{T}) = \sum_{r=0}^{3} \text{weight}(t^{(r)}) \leq B_\theta + B_\theta - 1 + (B_\theta - 1)^2 = B_\theta^2.$$

Figure 4.3 illustrates such a possible construction for the standard AES case. The active state bytes are marked with a blue colouring.

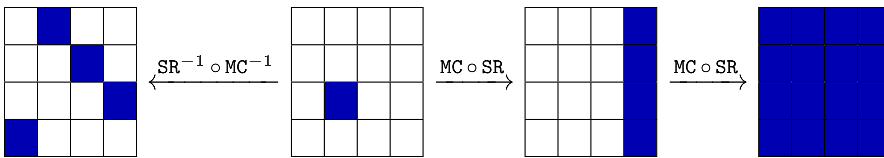

Figure 4.3: Optimal four-round trail in the AES using the standard `ShiftRows` layer.

Unfortunately, this argument does not work for the eight-round propagation and cubed state dimensions in order to prove an optimal value of B_θ^3 since the trail collapses at some point. It becomes not that trivial to construct such a trail as above for eight rounds. However, all experiments provide a number lower than this bound. Because of that observations, it is likely to assume this to be the optimal bound. If the state is not cubed but arbitrary, this is not true in general and there exist permutations which guarantee more than B_θ active bundles.

Example: Let the state be in dimension 3×13 and let $\sigma_1 = $ 0-4-8 and $\sigma_2 = $ 1-3-7. Then $(\sigma_1, \sigma_2)_8 \xrightarrow{4} 66$.

As a remark, the construction made above illustrates that for squared state dimensions and an optimal branch number, all diffusion-optimal permutations are equivalent. Diffusion-optimality in this case means that the permutation spreads the bundles within every column over all different columns. Given a diffusion-optimal permutation sequence, the four-round trail construction made above contains exactely B_θ^2 active S-boxes. Thereby the input pattern can be assumed to be the same as the output pattern. This results in an optimal $4n$-round trail weight of nB_θ^2. This construction can be made for all diffusion-optimal permutations which can be different depending on the round. Thus, diffusion-optimality implies an optimal guaranteed trail weight in the case of squared state dimensions and perfect column mixings. An arising question is whether the converse holds as well. Especially, can the standard AES be improved with a focus on the resulting eight-round trail weights using other permutations? It would be necessary that such a permutation sequence contains non-diffusion-optimal entries. By intuition, one could assume that this is not possible.

4.4.1 Considering Arbitrary Permutations

Since there exist $N!$ possible permutations for each row within the state, the iteration over all possibilities becomes infeasible very soon, even in the smallest dimension. Thus, the main focus lies on the special case of cyclic rotations. In addition, cyclic rotations are quite easy to implement.

In the following, the iteration over arbitrary permutations is done for the small dimension 2×4, if one unique layer is assumed for every round. In this case, $4!^2 = 576$ possibilites have to be considered. It turns out that the optimum of a guaranteed eight-round trail weight of 27 can not be reached using only one single layer.

Using the standard form, even a consideration of arbitrary two-alternating permutations is possible. Then the amount of possibilites raises to $24^3 = 13824$ which is quite easy to cope with for the small models. Listing A.6 in the appendix provides a list of all possibilities that reach the eight-round optimum for an optimal branch number. Equivalent variants may be left out.

4.4.2 Considering Two-Alternating Cyclic Rotations

In the following only cyclic rotations, thus permutations of the form $\sigma = s_1\text{-}\dots\text{-}s_M$ are considered. In particular, at most two-alternating shifting sequences $(\sigma_1, \sigma_2)_8$ are analyzed. Without loss of generality, we can assume that the offsets are in increasing order, thus $s_1 \leq s_2 \leq \cdots \leq s_M$ for each of the permutations. In addition, there is an interesting observation for sequences of cyclic shiftings. An element within the sequence can be substituted by its inverse, resulting to an equivalent sequence.

Lemma 4.4.1. *Let $\sigma_1, \dots, \sigma_R \in S_{M\times N}$ be cyclic rotations within the rows. Then the eqivalence $(\sigma_1, \dots, \sigma_r, \dots, \sigma_R)_\sim(\sigma_1, \dots, \sigma_r^{-1}, \dots, \sigma_R)$ holds for every round index r.*

Proof. For a shifting $\sigma_r = s_1\text{-}s_2\text{-}\dots\text{-}s_M$, the inverse can be expressed as

$$\sigma_r^{-1} = (-s_1 \mod M)\text{-}(-s_2 \mod M)\text{-}\dots\text{-}(-s_M \mod M).$$

Thus, it is the same permutation as a right-shift. By substituting the mixing layer $(\texttt{MC}_1^{(r)}, \ldots, \texttt{MC}_N^{(r)})$ by $(\texttt{MC}_N^{(r)}, \ldots, \texttt{MC}_1^{(r)})$ in any AES-like cipher, the problem is symmetric considering the shift direction. □

As an implication, one obtains

$$(\sigma_1, \ldots, \sigma_R) \sim (\sigma_1^{-1}, \ldots, \sigma_R^{-1}) \sim (\sigma_R, \ldots, \sigma_1),$$

meaning that the order of the shiftings can be reversed.

The standard form provided in Theorem 3.4.5 states that the first row within the first $k-1$ permutations in a k alternating structure can be assumed as unpermuted. Considering arbitrary permutations, the restriction to only the first $k-1$ rows is necessary. As an example, there are single permutations in $S_{2\times 4}$ which guarantee 21 active bundles over eight rounds, but none of them leaves the first row unpermuted. If one is interested in cyclic shiftings instead of arbitrary permutations, this restriction is not necessary any more. Even the kth rotation can now be assumed to be in that form. The argument is the same as in Lemma 3.4.4. Instead of using column swaps, one can consider shiftings of the whole state until the first row is unchanged. The difference is that these state shiftings commute with the cyclic rotations σ_r, resulting that σ_{k+i} can be changed to the σ_i again using only state shiftings. Now, one is able to extend the changes until σ_R is reached.

One of the questions is, whether it is possible to reach the optimum of B_θ^3 guaranteed S-boxes using two-alternating sequences of row shiftings. For the cases 2×4 and 3×9, the tool used in Section 4.3 is adapted in order to iterate over all possible sequences of that form. We use the consideration made above and the fact that $(\sigma_1, \sigma_2)_8 \sim (\sigma_1^{-1}, \sigma_2^{-1})_8$. It follows without loss of generality that it is sufficient to assume an increasing order of the s_i within each permutation, $s_1 = 0$ for σ_1, σ_2 because of the standard form (and the argument above) and $s_2 \leq \lfloor \frac{M}{2} \rfloor$ for σ_1, σ_2 because of the equivalence of the inverse. The modified code and a list of all the results in the lowest dimension is provided in the appendix. Unfortunately, a lot of models in the 4×16 case take much computation time even with the Gurobi solver so that this brute force approach is not done for this dimension anymore. In addition, the number of possibilities is rather high. Instead, just a few selected models are presented.

By this brute force method, the following observations turned out.

Fact 4.4.2. *For a 2×4 or a 3×9 state and optimal* `MixColumns` *branch number, there are no two-alternating cyclic rotations within the rows (σ_1, σ_2) that guarantee B_θ^3 active S-boxes over eight rounds.*

It may be possible that this observation holds for higher arrangements and that the optimum cannot be reached using two-alternating shiftings. Unfortunately, it cannot be shown in general in our experiments. For 2×4 the optimum is 24 and for 3×9 the optimum is 60, which is $B_\theta^3 - B_\theta$.

Within Table 4.4, some results are illustrated for the interesting case of a 4×16 state. It turns out that Gurobi may be significantly faster for the more complex cases, if the

`MipFocus` parameter is set to 1. This setting has more focus on finding smaller trail weights than on proving the optimitality of certain solutions. One can notice that the time needed for solving the models varies within a huge range depending on the amount of guaranteed active bundles. For models which can already be solved fast with the default setting, this approach takes some more time. However, the complex ones obtain a speed-up of up to ten times faster.

Because of the observations above, the optimality of 120 active bundles for any two-alternating shiftings is assumed.

Table 4.4: Mimimum number of active bundles in a 4×16 state for two-alternating shiftings.

Shifting σ_1	**Shifting** σ_2	**8-round minimum**	**Ex. time: Gurobi (`MipFocus` = 1)**
0-1-2-3	0-2-4-6	97	$\sim$ 1m 50s
0-1-2-3	0-3-6-9	116	$\sim$ 45m 40s
0-1-2-3	0-4-8-12	80	22.7s
0-1-2-3	0-2-7-9	120	$\sim$ 1h 14m 45s
0-1-2-3	0-1-5-8	115	$\sim$ 42m 10s
0-1-2-3	0-1-6-10	118	$\sim$ 54m 05s
0-1-2-3	0-1-6-11	116	$\sim$ 38m 20s
0-1-2-3	0-1-6-12	110	$\sim$ 11m 55s
0-1-2-3	0-1-10-12	106	$\sim$ 9m 30s
0-2-4-6	0-3-6-9	116	$\sim$ 36m 10s
0-2-4-6	0-4-8-12	61	5.8s
0-2-4-6	0-2-7-9	97	$\sim$ 1m 40s
0-2-4-6	0-1-5-8	105	$\sim$ 2m 20s
0-2-4-6	0-1-6-10	80	$\sim$ 40s
0-2-4-6	0-1-6-11	116	$\sim$ 38m 15s
0-2-4-6	0-1-6-12	80	$\sim$ 1m 25s
0-3-6-9	0-4-8-12	80	26.9s
0-3-6-9	0-2-7-9	116	$\sim$ 34m 45s
0-3-6-9	0-1-5-8	106	$\sim$ 5m 55s
0-3-6-9	0-1-6-10	90	$\sim$ 40s
0-3-6-9	0-1-6-11	120	$\sim$ 1h 23m 20s
0-3-6-9	0-1-6-12	110	$\sim$ 6m 25s
0-4-8-12	0-2-7-9	80	22.87s
0-4-8-12	0-1-5-8	61	14.81s
0-4-8-12	0-1-6-10	61	16.01s
0-4-8-12	0-1-6-11	80	25.31s
0-4-8-12	0-1-6-12	61	15.95s
0-1-6-10	σ_1	80	26.13s
0-1-6-11	σ_1	60	19.62s

There is an interesting fact one can notice in these results. There are guaranteed trail weights which are not a multiple of the branch number. Since the number of considered rounds is even (8), this implies that there exist permutation and MDS mixing layers such that the smallest non-trivial trail $\mathfrak{T}$ contains a two-round subtrail $\mathfrak{T}^{'}$ with weight($\mathfrak{T}^{'}$) $> B_\theta$. This is a surprising observation since one might assume by intuition that the lowest weight is realized by the smallest possible amount of active bundles for every two rounds, which is the `MixColumns` branch number. In other words, it is necessary for the algorithm to compare the sum of two consecutive columns with B_θ by $\geq$ instead of equality.

In addition, the results in the upper table look quite arbitrary and it is not possible to notice any obvious structure.

4.4.3 Reaching the Optimum with Cyclic Rotations

This section points out one interesting result. Since the optimum was not found for two-alternating shiftings, one can think that shiftings are not sufficient. However, there is an easy way to reach the optimal solution considering four-alternating cyclic rotations within the rows. Considering no alternating structure at all, there exist even more possibilities. After another brute force approach on the 2×4 state, the optimal shiftings are tried to adapt to the 4×16 arrangement.

Using 4-Alternating Shiftings

Table 4.5 presents the optimal solution reached with only the use of two shiftings and shows that the order is relevant now. Even a cyclic swapping of the used permutations does not reach the optimum. This realization using rotations within the rows is the result one wants to have since these rotations are rather easy to implement. Now it is not necessary to cope with the 120° or 90° cube turn anymore.

Table 4.5: Optimal bound in a 4×16 state with four-alternating shiftings.

σ_1	σ_2	σ_3	σ_4	**8-round minimum**	**Ex. time: Gurobi**
0-1-2-3	0-4-8-12	0-4-8-12	0-1-2-3	**125**	166.36s
0-1-2-3	0-1-2-3	0-4-8-12	0-4-8-12	110	94.12s
0-4-8-12	0-1-2-3	0-1-2-3	0-4-8-12	95	227.48s
0-4-8-12	0-4-8-12	0-1-2-3	0-1-2-3	110	125.40s
0-1-2-3	0-4-8-12	0-1-2-3	0-4-8-12	80	7.97s
0-1-2-3	0-4-8-12	0-4-8-12	0-4-8-12	50	2.23s
0-4-8-12	0-1-2-3	0-1-2-3	0-1-2-3	100	633.39s

This result contains a superbox structure as well. Thus, it can be proven quite easily for arbitrary cubed dimensions. This is done in the next chapter.

Using No Superbox Structure

For the most simple dimension, the optimal solution can also be reached using a more asymmetric sequence pattern which contains no alternating structure. Thus, it is unlikely that there exists such a hidden superbox argument. Using permutations without a superbox pattern would be a desirable method since there are some other vulnerabilites of superbox structures [BDPVA11][GP10]. The next chapter explains more about these structures.

Table 4.6 presents lots of these possibilites. Equivalent variants and the four-alternating solution are left out. Since the bound is independent of the first and the last permutation, they can be arbitrarily defined.

Table 4.6: Optimal non-alternating shiftings in a 2×4 state.

σ_2	σ_3	σ_4	σ_5	σ_6	σ_7	**8-round minimum**
0-1	0-1	0-1	0-1	0-1	0-2	**27**
0-1	0-1	0-1	0-1	0-2	0-2	**27**
0-2	0-1	0-1	0-1	0-1	0-1	**27**
0-2	0-1	0-1	0-1	0-1	0-2	**27**
0-2	0-2	0-1	0-1	0-1	0-1	**27**

Because of these results, it might be useful to consider no alternating structures for higher dimensions as well. The problem is the huge number of possibilities making this approach computationally infeasible for the interesting cases. Unfortunately, optimal non-alternating layers for the interesting dimension 4×16 or even for the 3×9 case are not found. To find such layers (or to prove their non-existence) is left as a question for future work.

5 Round Propagation

Round propagation is about proving bounds on the number of acitve S-boxes within a trail over a certain number of rounds. Using this formalized method, experimental results may be easily verified for some cases. Usually, this approach is applied during the designing process of a new cipher. In this chapter, the optimal four-alternating sequence developed in Chapter 4 is analyzed using an iterated superbox argument.

5.1 The Standard AES Case

During the design process of the AES, Daemen and Rijmen already proved a theorem which states that the minimum number of active S-boxes over four rounds is at least 25. The detailed proof is available in [DR02]. The basic principle is the knowledge of the behaviour over two consecutive rounds, e.g the lower bound of B_θ for a two-round trail weight. The four rounds are then considered as only two rounds using big S-boxes, so called superboxes. All of the following can be considered as an iteration of the whole process described in the literature.

5.2 Superboxes

The superbox argument is about iterating the S-box structure to a higher dimension and is widely-used in the security analysis of AES-derived primitives such as ECHO [BBG+09] or PAEQ [BK14]. It can be used in order to iterate the Four-Round Propagation Theorem and prove bounds for 8 or 16 rounds, if the permutation layer contains such a superbox structure. The main idea is to consider one whole column operation within certain rounds as a big substitution step. For a superbox argument, it is important that the whole state splits into distinct instances such that the cipher operates on these states independently. This approach is already sketched in the literature in order to do the four-round propagation. Furthermore, [DR06] provides a good intuitive definition which we will use. In the following, the AES transformations are denoted in a shorter form for better reading. Also, the concrete mixings are left out. For our purposes, the superbox is intuitively defined as follows.

Definition 5.2.1. Given an AES-like cipher, the two-round *superbox* under the round key K is defined as $\mathtt{SSB}_K = \mathtt{SB} \circ \mathtt{AddK}_K \circ \mathtt{MC} \circ \mathtt{SB}$.

If the state dimension is $M \times M^2$ and the used permutations allow to separate the state into distinct instances, an *iterated four-round superbox (megabox)* can be defined as

$$\sigma\text{-}\mathtt{MSB}_{K_1,K_2,K_3} = \mathtt{SSB}_{K_3} \circ \mathtt{P}_\sigma \circ \mathtt{AddK}_{K_2} \circ \mathtt{MC} \circ \mathtt{P}_\sigma \circ \mathtt{SSB}_{K_1}.$$

Thereby, the cubed state decomposes into M distinct four-round megaboxes. This may also be iterated again in order to define gigaboxes, etc.

One can notice that this definition of the (standard) superbox covers exactely all substitution steps over two rounds of one column during an execution of the AES. Thereby the input are the four bytes which will be shifted into the column by the `ShiftRows` operation. To illustrate this, two consecutive AES rounds are computed as

$$\texttt{AddK} \circ \texttt{MC} \circ \texttt{SR} \circ \texttt{SB} \circ \texttt{AddK} \circ \texttt{MC} \circ \texttt{SR} \circ \texttt{SB}\,.$$

Since the bytewise permutation commutes with the substitution layer, these two rounds can be written with the use of superboxes as $\texttt{AddK} \circ \texttt{MC} \circ \texttt{SR} \circ \texttt{SSB} \circ \texttt{SR}$. Thus, the superbox covers all the substitution steps over two rounds. [GP10] also describes this view in detail.

This approach can be used for multi-round propagation since one can talk about active superboxes instead of active S-boxes. Every active superbox must contain at least B_θ active S-boxes since it covers two rounds. The principle is to consider distinct instances of substates on which the superboxes operate in an independent way.

Analogous, the megaboxes cover four rounds on a squared instance. An important assumption in order to define the megabox is the fact that for a concrete cipher the permutation layer allows to separate the state into distinct ones, such that the megabox structure holds for all of the considered number of rounds.

5.3 Eight-Round Propagation with Iterated Superboxes

This generalization to megaboxes proves the eight-round bound of 125 active S-boxes using the four-alternating permutation sequence. For a better understanding, the proof again illustrates in detail how to apply this concept.

Theorem 5.3.1 (Eight-Round Propagation Theorem)**.** *Let $\mathfrak{C}$ be an AES-like cipher with state dimension $M \times M^2$ using the 4 alternating shiftings*

$$\sigma_1 = 0\text{-}1\text{-}2\text{-}\ldots\text{-}(M-1) \quad \sigma_2 = 0\text{-}M\text{-}2M\text{-}\ldots\text{-}(M-1)M \quad \sigma_3 = \sigma_2 \quad \sigma_4 = \sigma_1.$$

Any non-trivial eight-round trail in $\mathfrak{C}$ starting with shifting σ_1 has at least B_θ^3 active S-boxes.

Proof. The considered eight rounds in the cipher can be experessed as

$$\begin{array}{lll}
\texttt{AddK}_{K_8} \circ \texttt{MC} \circ \texttt{P}_{\sigma_1} \circ \texttt{SB} & \circ & \texttt{AddK}_{K_7} \circ \texttt{MC} \circ \texttt{P}_{\sigma_2} \circ \texttt{SB} \quad \circ \\
\texttt{AddK}_{K_6} \circ \texttt{MC} \circ \texttt{P}_{\sigma_2} \circ \texttt{SB} & \circ & \texttt{AddK}_{K_5} \circ \texttt{MC} \circ \texttt{P}_{\sigma_1} \circ \texttt{SB} \quad \circ \\
\texttt{AddK}_{K_4} \circ \texttt{MC} \circ \texttt{P}_{\sigma_1} \circ \texttt{SB} & \circ & \texttt{AddK}_{K_3} \circ \texttt{MC} \circ \texttt{P}_{\sigma_2} \circ \texttt{SB} \quad \circ \\
\texttt{AddK}_{K_2} \circ \texttt{MC} \circ \texttt{P}_{\sigma_2} \circ \texttt{SB} & \circ & \texttt{AddK}_{K_1} \circ \texttt{MC} \circ \texttt{P}_{\sigma_1} \circ \texttt{SB}
\end{array}$$

Without loss of generality, one can assume that only one mixing is used for every column in every round. Since the `SubBytes` step operates on the bundles separately, it commutes with the `Permute` layer and the following expression is equal to the one above.

$$\begin{array}{llll} \mathtt{AddK}_{K_8} \circ \mathtt{MC} \circ \mathtt{P}_{\sigma_1} \circ \mathtt{SB} & \circ & \mathtt{AddK}_{K_7} \circ \mathtt{MC} \circ \mathtt{SB} \circ \mathtt{P}_{\sigma_2} & \circ \\ \mathtt{AddK}_{K_6} \circ \mathtt{MC} \circ \mathtt{P}_{\sigma_2} \circ \mathtt{SB} & \circ & \mathtt{AddK}_{K_5} \circ \mathtt{MC} \circ \mathtt{SB} \circ \mathtt{P}_{\sigma_1} & \circ \\ \mathtt{AddK}_{K_4} \circ \mathtt{MC} \circ \mathtt{P}_{\sigma_1} \circ \mathtt{SB} & \circ & \mathtt{AddK}_{K_3} \circ \mathtt{MC} \circ \mathtt{SB} \circ \mathtt{P}_{\sigma_2} & \circ \\ \mathtt{AddK}_{K_2} \circ \mathtt{MC} \circ \mathtt{P}_{\sigma_2} \circ \mathtt{SB} & \circ & \mathtt{AddK}_{K_1} \circ \mathtt{MC} \circ \mathtt{SB} \circ \mathtt{P}_{\sigma_1} & \end{array}$$

Now, one can define a `MegaSubBytes` step (`MSB`) as

$$\sigma_2\text{-}\mathtt{MSB}_{K',K'',K'''} = \mathtt{SB} \circ \mathtt{AddK}_{K'''} \circ \mathtt{MC} \circ \mathtt{SB} \circ \mathtt{P}_{\sigma_2} \circ \mathtt{AddK}_{K''} \circ \mathtt{MC} \circ \mathtt{P}_{\sigma_2} \circ \mathtt{SB} \circ \mathtt{AddK}_{K'} \circ \mathtt{MC} \circ \mathtt{SB}$$

which depends on the three round keys K', K'', K'''. This megabox can be understood as a huge S-box operating on M^2 bundles. Figure 5.1 illustrates where the megaboxes lie in the example of a 4×16 state. Using this iterated superbox approach, the eight rounds

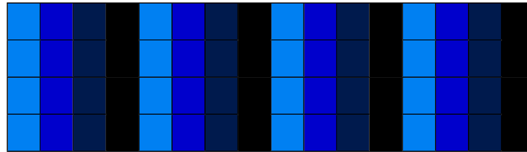

Figure 5.1: Position of the 4 megaboxes in a 4×16 state.

can be expressed as

$$\mathtt{AddK}_{K_8} \circ \mathtt{MC} \circ \mathtt{P}_{\sigma_1} \circ \mathtt{MSB}_{K_5,K_6,K_7} \circ \mathtt{P}_{\sigma_1} \circ \mathtt{AddK}_{K_4} \circ \mathtt{MC} \circ \mathtt{P}_{\sigma_1} \circ \mathtt{MSB}_{K_1,K_2,K_3} \circ \mathtt{P}_{\sigma_1} .$$

One can notice that one megabox represents an independent $M \times M$ AES-like state, using a permutation with optimal diffusion power operating on this state. Since any four-round trail weight is independent from the first round permutation and independent from the last round, the number of active S-boxes in one megabox is exactely the number of active S-boxes in the correspnding four-round trail in the $M \times M$ cipher. Thus, the Four-Round Propagation Theorem can be applied and it follows that one megabox contains at least B_θ^2 active S-boxes (supposed if one is active at all). It is left to show that the linear transformation step

$$\mathtt{P}_{\sigma_1} \circ \mathtt{AddK}_{K_4} \circ \mathtt{MC} \circ \mathtt{P}_{\sigma_1}$$

has a branch number of at least B_θ with respect to megaboxes, which means that the (non-trivial) sum of the active megaboxes before and after this transformation is at least B_θ.

Assume that at least one megabox is active. After applying $\mathtt{P}_{\sigma_1}$, every column obtains an input bundle from every megabox state, as Figure 5.2 illustrates. Because of the assumption, there exists an active column n. After applying `MC`, this column has an (input/output) sum of at least B_θ active S-boxes. Since $\mathtt{P}_{\sigma_1}$ shifts all bundles in n to different megabox states again, there are at least B_θ active megaboxes.

□

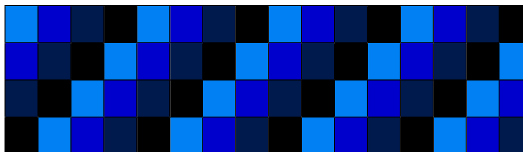

Figure 5.2: Mixing of megaboxes after applying $\mathtt{P}_{\sigma_1}$.

The last step is using the fact that $\mathtt{P}_{\sigma_1}$ keeps the megabox structure. This is exactely what is meant by allowing to seperate the state into distinct instances in order to define an iterated superbox.

5.4 Disadvantages

Although a superbox structure is quite easy to handle, there are several vulnerabilities against it. Worth mentioning are rebound attacks due to strong alignment, which might especially be a problem for AES-like hash function designs [BDPVA11][GP10]. Thus, it might be better to use for example one of the strong sequences analyzed in Table 4.4, which contain 118 or 120 acitve bundles. An amount of 120 active S-boxes can not contain such a superbox pattern and the difference of only five S-boxes does not often play a major role. As long as there are no optimal alternatives known, these permutations might be a better choice. It depends on the concrete application and conditions one wants to achieve. Of course, some other aspects should be kept in mind as well, such as diffusion properties of the used permutations. Understanding non-superbox permutation structures can be considered a major topic for future work.

6 Conclusion

This final chapter provides a summary and an evaluation of the results developed during this work. It also gives a short survey on possible future work.

6.1 Summary

After providing a formal description of the problem, we proved certain equivalences of different permutation sequences in order to reduce the problem domain. It turned out that it is enough to concentrate on permutations within the rows, if the concrete mixing and substituiton layers are not known and thus be considered as a black box.

In the next step, an experimental analysis of some interesting sequences was done. We explained the concept of mixed-integer linear programming and showed the tightness of the determined lower bound in the case of optimal MDS mixings. Surprisingly, we noticed that multi-round trails behave different from two-round trails, since a minimum-weighted trail can contain two-round subtrails which do not necessarily have to be minimum weighted. One goal of the experiments was to find cyclic shiftings which guarantee the expected optimal lower bound of B_θ^3 active S-boxes over eight round for cubed states. For small dimensions, it turned out that it is not possible to reach this bound using only two-alternating shiftings. The general case of arbitrary state dimensions is left as an open question.

As one of the main results we finally provided a permutation layer fulfilling this eight-round bound. It uses only two different cyclic shiftings. Since the order of them is reversed every second time, this resulted in a four-alternating rotation layer. Thereby, a superbox structure was considered to be inherent within the layer.

6.2 Future Work

As already stated in several sections within this thesis, there are some conjectures left to show. One of these questions is whether a guaranteed eight-round trail weight of B_θ^3 for cubed states using an optimal branch number B_θ is the optimal value possible to reach. A similar question is the relationship between diffusion properties of permutation layers and their resulting trail-weights. Another important topic would be to search for an alternative permutation layer for the interesting dimensions which guarantees the same bound as the superbox approach. More generally, an analysis of non-superbox structures and obtaining measures on how close a specific structure contains a superbox argument would be of huge interest. Combining these results with the consideration on diffusion properties would also be worth to consider.

Since we mainly concentrated on cubed state dimensions, other arrangements might be the topic of further analysis as well in order to deepen the understanding of permutation layers and their resulting trail weights.

A Listings of the MILP Tools

In the following, the listing of the tools used in the experiments and some solutions are given. At first, the case of examining a specific permutation layer is presented.

```
#include <stdio.h>
#include <stdlib.h>
#include "gurobi_c.h"           //gurobi solver is used

// Include the file with the specific problem dimensions and permutations
//#include "2x4solver.h"
//#include "3x9solver.h"
#include "4x16solver.h"

// switch off presolving, if error occurs for some models
//#define NOT_PRESOLVE

// storage for state indices and counter for round and index of dummyvariables
int state[ROUNDS][ROWS][COLUMNS]; // state matrix (containing only indices)
int dummyindex;                 // next index for new dummy variables
int actualRound;                // the current round (0 - ROUNDS-1)

// loop counters
int row, column, i;

// global variables for gurobi solver
double   obj[(ROUNDS+1)*ROWS*COLUMNS];  // contains the objective coefficient of the state
     variables over all rounds (will be all 1)
char     vtype[(ROUNDS+1)*ROWS*COLUMNS]; // type of the variables (state + dummy variables) (will
     later defined as integers >= 0)
int      varindex[ROUNDS*ROWS*COLUMNS+1]; // saves indices of the variables in the constraint
double   varcoeff[ROUNDS*ROWS*COLUMNS+1]; // saves coefficients of the variables in the
     constraints
double   sboxes[ROUNDS*ROWS*COLUMNS];   // saves the solution pattern
// error variable; 1 if error, 0 otherwise
int error = 0;

// Permute the state by PI[permnmb]
void Permute(int permnmb)
{
    // filling statematrix with permutated indices (depending on actual round)
    for (row = 0; row < ROWS; row++)
    {
        for (column = 0; column < COLUMNS; column++)
        {
            state[actualRound][row][column] = PI[permnmb][row][column] + ROWS * COLUMNS *
                 actualRound;
        }
    }
}

// Creates the constraints of the MixColumn layer depending on the branch number and finishes
     actual round; returns 0 if succesful, 1 otherwise
int MixColumns(GRBmodel* model)
```

```
{
    for (column = 0; column < COLUMNS; column++)
    {
        // Adding constraint for the current column in current round
        for (row = 0; row < ROWS; row++)
        {
            varindex[row] = state[actualRound][row][column];
            varindex[row+ROWS] = row + ROWS * column + ROWS * COLUMNS * (actualRound+1);
            varcoeff[row] = 1;
            varcoeff[row+ROWS] = 1;
        }
        // considereing dummy
        varindex[2*ROWS] = (ROUNDS+1)*ROWS*COLUMNS + dummyindex;
        varcoeff[2*ROWS] = (-1)*BRANCHNMB;
        // Finally add constraint
        error = GRBaddconstr(model, 2*ROWS+1, varindex, varcoeff, GRB_GREATER_EQUAL, 0.0, NULL);
        if (error)
            return 1;

        // dummyvariable is needed to consider trivial case with no active sboxes in the column or
             non-optimal branch numbers
        // add these constraints
        for (row = 0; row < ROWS; row++)
        {
            // checking column for active s-boxes with these constraints
            varindex[0] = (ROUNDS+1)*ROWS*COLUMNS + dummyindex;
            varindex[1] = state[actualRound][row][column];
            varcoeff[0] = 1;
            varcoeff[1] = -1;
            error = GRBaddconstr(model, 2, varindex, varcoeff, GRB_GREATER_EQUAL, 0.0, NULL);
            if (error)
                return 1;

            varindex[0] = (ROUNDS+1)*ROWS*COLUMNS + dummyindex;
            varindex[1] = row + ROWS * column + ROWS * COLUMNS * (actualRound+1);
            varcoeff[0] = 1;
            varcoeff[1] = -1;
            error = GRBaddconstr(model, 2, varindex, varcoeff, GRB_GREATER_EQUAL, 0.0, NULL);
            if (error)
                return 1;
        }

        // if branch number is not optimal (<ROWS+1), make sure that an active column is not turned
              non-active
        if(BRANCHNMB <= ROWS)
        {
            for (row = 0; row < ROWS; row++)
            {
                varindex[row] = state[actualRound][row][column];
                varcoeff[row] = 1;
            }
            varindex[ROWS] = (ROUNDS+1)*ROWS*COLUMNS + dummyindex;
            varcoeff[ROWS] = -1;
            error = GRBaddconstr(model, ROWS+1, varindex, varcoeff, GRB_GREATER_EQUAL, 0.0, NULL);
            if (error)
                return 1;

            for (row = 0; row < ROWS; row++)
            {
                varindex[row] = row + ROWS * column + ROWS * COLUMNS * (actualRound+1);
                varcoeff[row] = 1;
            }
```

```
            varindex[ROWS] = (ROUNDS+1)*ROWS*COLUMNS + dummyindex;
            varcoeff[ROWS] = -1;
            error = GRBaddconstr(model, ROWS+1, varindex, varcoeff, GRB_GREATER_EQUAL, 0.0, NULL);
            if (error)
                return 1;
        }

        // a new dummy variable for each column
        dummyindex++;
    }
    // finish the round
    actualRound++;

    return 0;
}

// Creates the MILPmodel of the SBox minimization problem depending on the permutations. Returns
     the model (NULL if error occured)
GRBmodel* create_MILPmodel(GRBenv* env)
{
    // reset dummy and round counter for new model
    dummyindex = 0;
    actualRound= 0;

    // create the new model (empty model)
    GRBmodel* model = NULL;
    error = GRBnewmodel(env, &model, NULL, 0, NULL, NULL, NULL, NULL, NULL);
    if (error)
        return NULL;

    // Now add the state variables to the model
    for (i=0; i<ROUNDS*ROWS*COLUMNS; i++)
    {
        // The objective coefficient of the state variables is 1, since the objective function is a
             [0]+a[1]+...+a[ROUNDS*ROWS*COLUMNS]
        obj[i] = 1;
        vtype[i] = GRB_INTEGER;
    }
    // Last state not in objective, set coefficitents to 0
    for (i=ROUNDS*ROWS*COLUMNS; i<(ROUNDS+1)*ROWS*COLUMNS; i++)
    {
        obj[i] = 0;
        vtype[i] = GRB_INTEGER;
    }
    // add these objective variables; all are non-negative integers (minimization by default)
    error = GRBaddvars(model, (ROUNDS+1)*ROWS*COLUMNS, 0, NULL, NULL, NULL, obj, NULL, NULL, vtype,
          NULL);
    if (error)
        return NULL;

    // Now add all dummy variables to the model (are not in objective function; obective
         coefficients are 0)
    // There will be a dummy variable for each column over all rounds (ROUNDS*COLUMNS
         dummyvariables)
    error = GRBaddvars(model, ROUNDS*COLUMNS, 0, NULL, NULL, NULL, NULL, NULL, NULL, vtype, NULL);
    if (error)
        return NULL;

    // Update model such that the new variables are integrated
    error = GRBupdatemodel(model);
    if (error)
        return NULL;
```

```
    // Simulate the cipher over ROUNDS rounds. Within the MixColumns layer, the constraints for
         the MILPmodel will be added
    for (i=0; i<ROUNDS; i++)
    {
        // permute by the actual permutation
        Permute(i % ALTERNATING);
        // Simulate mixing layer and add constraints
        error = MixColumns(model);
        if (error)
            return NULL;
    }

    // excluding the trivial solution with no active sboxes at all
    for (i=0; i < ROWS * COLUMNS * ROUNDS; i++)
    {
        varindex[i] = i;
        varcoeff[i] = 1;
    }
    // add the constraint
    error = GRBaddconstr(model, ROUNDS*ROWS*COLUMNS, varindex, varcoeff, GRB_GREATER_EQUAL, 1.0,
         NULL);
    if (error)
        return NULL;

    return model;
}

// Solves a MILPmodel and returns the optimized objective value; returns 0 if error occured or
     model is not solved optimal
int solve_MILPmodel(GRBmodel* model)
{
    double solution;
    int status;

    // change the mipfocus to 1 if desired
    #ifdef CHANGE_MIPFOCUS
    error = GRBsetintparam(GRBgetenv(model), GRB_INT_PAR_MIPFOCUS, 1);
    if (error)
        return 0;
    #endif

    // solve
    error = GRBoptimize(model);
    if (error)
        return 0;

    // check if model is solved optimal
    error = GRBgetintattr(model, GRB_INT_ATTR_STATUS, &status);
    if (error || (status != GRB_OPTIMAL))
        return 0;

    // get the solution
    error = GRBgetdblattr(model, GRB_DBL_ATTR_OBJVAL, &solution);
    if (error)
        return 0;

    // get the pattern
    error = GRBgetdblattrarray(model, GRB_DBL_ATTR_X, 0, ROWS*COLUMNS*ROUNDS, sboxes);
    if (error)
        return 0;
```

```
    // for integer cast
    solution = solution + 0.3;

    return (int)solution;
}

int main()
{
  // Variables for the solution
  int activeSBoxes;
  double running_time;

  // File to print the solution in
  FILE *fp;
  fp = fopen("solution.txt", "w");

  // Create gurobi environment
  GRBenv *environment = NULL;
  error = GRBloadenv(&environment, "guaranteed_trail_weight.log");
  if (error || environment == NULL)
    return 1;

  //switch off presolve, if error occurs
  #ifdef NOT_PRESOLVE
    error = GRBsetintparam(environment, "PRESOLVE", 0);
    if (error)
        return 1;
  #endif

  // Create an empty MILPModel which will handle the minimization problem
  GRBmodel* guaranteed_trail_weight = NULL;

  // Now simulate the permutation layer and solve the model
  fprintf(fp, "Minimize active s-boxes in a %i x %i state over %i rounds (branchnumber %i)\n\n",
       ROWS, COLUMNS, ROUNDS, BRANCHNMB);
  guaranteed_trail_weight = create_MILPmodel(environment);
  activeSBoxes = solve_MILPmodel(guaranteed_trail_weight);

  // Write the model to file
  error = GRBwrite(guaranteed_trail_weight, "guaranteed_trail_weight.lp");
  if (error)
    return 1;

  // Print solution in file and check if error occured
  if (!activeSBoxes)
    fprintf(fp, "Error testing shifting!");
  else
  {
    fprintf(fp, "SBoxes: %i\n\n", activeSBoxes);
    fprintf(fp, "The pattern is: \n \n");
    for (i=0; i<ROUNDS; i++)
    {
        fprintf(fp, "round %i:\n", i);
        for (row=0; row<ROWS; row++)
        {
            fprintf(fp, "%i", (int)sboxes[row+ROWS*COLUMNS*i]);
            for (column=1; column<COLUMNS; column++)
            {
                fprintf(fp, ", %i", (int)sboxes[row+ROWS*column+ROWS*COLUMNS*i]);
            }
            fprintf(fp, "\n");
        }
```

```
        fprintf(fp, "\n");
    }
    error = GRBgetdblattr(guaranteed_trail_weight, GRB_DBL_ATTR_RUNTIME, &running_time);
    if (error)
        return 1;
    fprintf(fp, "\nSolution was found in %f seconds", running_time);
    }

  // Free the model and the environment
  GRBfreemodel(guaranteed_trail_weight);
  GRBfreeenv(environment);

  return 0;
}
```

Listing A.1: Mainfile for determining the guaranteed trail weight of a specific permutation layer

To illustrate, how the dimension and the permutation layer is implemented, the header "4x16solver.h" is given as an example.

```
// define cipher structure here
// ...............................................................................
#define ROWS 4
#define COLUMNS 16              // dimensions of the state matrix
#define ROUNDS 8                // number of rounds to be considered

// For higher dimensions, like 4 x 16, the solver is much faster, if the mipfocus parameter is
     changed to 1
#define CHANGE_MIPFOCUS

#define BRANCHNMB 5             // the differential branch number of the MixColumn layer

#define ALTERNATING 2           // When starts the alternating of permutations
                                // (example: if ALTERNATING = 2, then only PI_1 and PI_2 is used
                                     in an alternating way)

// permutations instead of ShiftRows, define them here, row by row:
// example: 4x4 AES-state and regular ShiftRows layer:
// 0, 4, 8, 12
// 5, 9, 13, 1
// 10, 14,2, 6
// 15, 3, 7, 11
const int PI[ALTERNATING][ROWS][COLUMNS] = {

    {{0, 4, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60}, // SR          (PI_1)
     {5, 9, 13, 1, 21, 25, 29, 17, 37, 41, 45, 33, 53, 57, 61, 49},
     {10, 14, 2, 6, 26, 30, 18, 22, 42, 46, 34, 38, 58, 62, 50, 54},
     {15, 3, 7, 11, 31, 19, 23, 27, 47, 35, 39, 43, 63, 51, 55, 59,}},

    {{48, 32, 16, 0, 52, 36, 20, 4, 56, 40, 24, 8,  60, 44, 28, 12},  //SR o rot(90) (PI_2)
     {33, 17, 1, 49, 37, 21, 5, 53, 41, 25, 9, 57,  45, 29, 13, 61},
     {18, 2, 50, 34, 22, 6, 54, 38, 26, 10, 58, 42, 30, 14, 62, 46},
     {3, 51, 35, 19, 7, 55, 39, 23, 11, 59, 43, 27, 15, 63, 47, 31}},

      };
// ...............................................................................
// end of definitions

/* some other examples for permutations:
```

```
    {{48, 32, 16, 0, 52, 36, 20, 4, 56, 40, 24, 8, 60, 44, 28, 12}, //rot(90)
     {49, 33, 17, 1, 53, 37, 21, 5, 57, 41, 25, 9, 61, 45, 29, 13},
     {50, 34, 18, 2, 54, 38, 22, 6, 58, 42, 26, 10, 62, 46, 30, 14},
     {51, 35, 19, 3, 55, 39, 23, 7, 59, 43, 27, 11, 63, 47, 31, 15}},

    {{60, 44, 28, 12, 61, 45, 29, 13, 62, 46, 30, 14, 63, 47, 31, 15}, //rot(120)
     {56, 40, 24, 8, 57, 41, 25, 9, 58, 42, 26, 10, 59, 43, 27, 11},
     {52, 36, 20, 4, 53, 37, 21, 5, 54, 38, 22, 6, 55, 39, 23, 7},
     {48, 32, 16, 0, 49, 33, 17, 1, 50, 34, 18, 2, 51, 35, 19, 3}},

     {{0, 4, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 52, 56, 60}, // 0-4-8-12
     {17, 21, 25, 29, 33, 37, 41, 45, 49, 53, 57, 61, 1, 5, 9, 13},
     {34, 38, 42, 46, 50, 54, 58, 62, 2, 6, 10, 14, 18, 22, 26, 30},
     {51, 55, 59, 63, 3, 7, 11, 15, 19, 23, 27, 31, 35, 39, 43, 47}},

    {{48, 32, 16, 0, 52, 36, 20, 4, 56, 40, 24, 8, 60, 44, 28, 12}, //0-4-8-12 o rot(90)
     {53, 37, 21, 5, 57, 41, 25, 9, 61, 45, 29, 13, 49, 33, 17, 1},
     {58, 42, 26, 10, 62, 46, 30, 14, 50, 34, 18, 2, 54, 38, 22, 6},
     {63, 47, 31, 15, 51, 35, 19, 3, 55, 39, 23, 7, 59, 43, 27, 11}},
*/
```

Listing A.2: 4x16solver.h with an example permutation layer

```
Minimize active s-boxes in a 4 x 16 state over 8 rounds (branchnumber 5)

SBoxes: 125

The pattern is:

round 0:
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1

round 1:
0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0
0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0
0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1
1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0

round 2:
1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

round 3:
1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0
1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0
1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0
1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0

round 4:
0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

round 5:
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

```
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

round 6:
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0

round 7:
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1


Solution was found in 17.568176 seconds
```

Listing A.3: Solution and trail pattern for the given model

Modified version in order to iterate over all possible k-alternating cyclic rotations in a $2 \times x$ state:

```
#include <stdio.h>
#include <stdlib.h>
#include "gurobi_c.h"           // gurobi solver is used

// switch off presolving, if error occures for some models
//#define NOT_PRESOLVING

// Structure of the 2 x ? state and number of rounds and MixColumns branch number. Adjust theses
     values.
#define COLUMNS 4               // dimensions of the state matrix
#define ROUNDS 8                // number of rounds to be considered
#define BRANCHNMB 3             // the differential branch number of the MixColumn layer

// This value cannot be adjusted (number of rows is fixed to 2)
#define ROWS 2

// define which sequences will be considered
#define SINGLE_ROTATIONS
#define TWO_ALTERNATING
//#define FOUR_ALTERNATING
//#define EIGHT_ALTERNATING       // full possibilities for eight-round-trail

int state[ROUNDS][ROWS][COLUMNS]; // state matrix (containing only indices)
int dummyindex;                 // next index for new dummy variables
int actualRound;                // the current round (0 - ROUNDS-1)

// loop counters
int row, column, i;

// global variables for gurobi solver
double   obj[(ROUNDS+1)*ROWS*COLUMNS];  // contains the objective coefficient of the state
     variables over all rounds (will be all 1)
char     vtype[(ROUNDS+1)*ROWS*COLUMNS]; // type of the variables (state + dummy variables) (will
     later defined as integers >= 0)
int      varindex[ROUNDS*ROWS*COLUMNS+1]; // saves indices of the variables in the constraint
double   varcoeff[ROUNDS*ROWS*COLUMNS+1]; // saves coefficients of the variables in the
     constraints
// error variable; 1 if error, 0 otherwise
int error = 0;
```

```
// Cyclic rotation within the rows of the state. Row1 is leftrotated by r1, etc...
void Rotate(int r1, int r2)
{
    // filling statematrix with permuted indices (depending on actual round)
    // first row:
    for (column = 0; column < COLUMNS; column++)
    {
        state[actualRound][0][column] = ((r1*ROWS+ROWS*(column)) % (ROWS*COLUMNS)) + ROWS * COLUMNS
             * actualRound;
    }
    // second row:
    for (column = 0; column < COLUMNS; column++)
    {
        state[actualRound][1][column] = ((r2*ROWS+ROWS*(column)+1) % (ROWS*COLUMNS)) + ROWS *
            COLUMNS * actualRound;
    }
}

// Creates the constraints of the MixColumn layer depending on the branch number and finishes
     actual round; returns 0 if succesful, otherwise 1
int MixColumns(GRBmodel* model)
{
    for (column = 0; column < COLUMNS; column++)
    {
        // Adding constraint for the current column in current round
        for (row = 0; row < ROWS; row++)
        {
            varindex[row] = state[actualRound][row][column];
            varindex[row+ROWS] = row + ROWS * column + ROWS * COLUMNS * (actualRound+1);
            varcoeff[row] = 1;
            varcoeff[row+ROWS] = 1;
        }
        // considering dummy
        varindex[2*ROWS] = (ROUNDS+1)*ROWS*COLUMNS + dummyindex;
        varcoeff[2*ROWS] = (-1)*BRANCHNMB;
        // Finally add constraint
        error = GRBaddconstr(model, 2*ROWS+1, varindex, varcoeff, GRB_GREATER_EQUAL, 0.0, NULL);
        if (error)
            return 1;

        // dummyvariable is needed to consider trivial case with no active sboxes in the column or
             non-optimal branch numbers
        // add these constraints
        for (row = 0; row < ROWS; row++)
        {
            // checking column for active s-boxes with these constraints
            varindex[0] = (ROUNDS+1)*ROWS*COLUMNS + dummyindex;
            varindex[1] = state[actualRound][row][column];
            varcoeff[0] = 1;
            varcoeff[1] = -1;
            error = GRBaddconstr(model, 2, varindex, varcoeff, GRB_GREATER_EQUAL, 0.0, NULL);
            if (error)
                return 1;

            varindex[0] = (ROUNDS+1)*ROWS*COLUMNS + dummyindex;
            varindex[1] = row + ROWS * column + ROWS * COLUMNS * (actualRound+1);
            varcoeff[0] = 1;
            varcoeff[1] = -1;
            error = GRBaddconstr(model, 2, varindex, varcoeff, GRB_GREATER_EQUAL, 0.0, NULL);
            if (error)
                return 1;
```

```
        }

        // if branch number is not optimal (<ROWS+1), make sure that an active column is not turned
              non-active
        if(BRANCHNMB <= ROWS)
        {
            for (row = 0; row < ROWS; row++)
            {
                varindex[row] = state[actualRound][row][column];
                varcoeff[row] = 1;
            }
            varindex[ROWS] = (ROUNDS+1)*ROWS*COLUMNS + dummyindex;
            varcoeff[ROWS] = -1;
            error = GRBaddconstr(model, ROWS+1, varindex, varcoeff, GRB_GREATER_EQUAL, 0.0, NULL);
            if (error)
                return 1;

            for (row = 0; row < ROWS; row++)
            {
                varindex[row] = row + ROWS * column + ROWS * COLUMNS * (actualRound+1);
                varcoeff[row] = 1;
            }
            varindex[ROWS] = (ROUNDS+1)*ROWS*COLUMNS + dummyindex;
            varcoeff[ROWS] = -1;
            error = GRBaddconstr(model, ROWS+1, varindex, varcoeff, GRB_GREATER_EQUAL, 0.0, NULL);
            if (error)
                return 1;
        }

        // a new dummy variable for each column
        dummyindex++;
    }
    // finish the round
    actualRound++;

    return 0;
}

// Creates the MILPmodel of the SBox minimization problem depending on the eight shiftings.
     Returns the model (NULL if error occured)
GRBmodel* create_MILPmodel(GRBenv* env, int s1_r1, int s1_r2, int s2_r1, int s2_r2, int s3_r1, int
      s3_r2, int s4_r1, int s4_r2,
    int s5_r1, int s5_r2, int s6_r1, int s6_r2, int s7_r1, int s7_r2, int s8_r1, int s8_r2)
{
    // reset dummy and round counter for new model
    dummyindex = 0;
    actualRound= 0;

    // create the new model (empty model)
    GRBmodel* model = NULL;
    error = GRBnewmodel(env, &model, NULL, 0, NULL, NULL, NULL, NULL, NULL);
    if (error)
        return NULL;

    // Now add the state variables to the model
    for (i=0; i<ROUNDS*ROWS*COLUMNS; i++)
    {
        // The objective coefficient of the state variables is 1, since the objective function is a
             [0]+a[1]+...+a[ROUNDS*ROWS*COLUMNS]
        obj[i] = 1;
        vtype[i] = GRB_INTEGER;
    }
```

```
// Last state not in objective, set coefficitents to 0
for (i=ROUNDS*ROWS*COLUMNS; i<(ROUNDS+1)*ROWS*COLUMNS; i++)
{
    obj[i] = 0;
    vtype[i] = GRB_INTEGER;
}
// add these objective variables; all are non-negative integers (minimization by default)
error = GRBaddvars(model, (ROUNDS+1)*ROWS*COLUMNS, 0, NULL, NULL, NULL, obj, NULL, NULL, vtype,
      NULL);
if (error)
    return NULL;

// Now add all dummy variables to the model (are not in objective function; obective
     coefficients are 0)
// There will be a dummy variable for each column over all rounds (ROUNDS*COLUMNS
     dummyvariables)
error = GRBaddvars(model, ROUNDS*COLUMNS, 0, NULL, NULL, NULL, NULL, NULL, NULL, vtype, NULL);
if (error)
    return NULL;

// Update model such that the new variables are integrated
error = GRBupdatemodel(model);
if (error)
    return NULL;

// Simulate the cipher over ROUNDS rounds. Within the MixColumns layer, the constraints for
     the MILPmodel will be added
for (i=0; i<ROUNDS; i++)
{
    // do the shiftings depending on the round
    if ((i % 8) == 0)
        Rotate(s1_r1, s1_r2);
    if ((i % 8) == 1)
        Rotate(s2_r1, s2_r2);
    if ((i % 8) == 2)
        Rotate(s3_r1, s3_r2);
    if ((i % 8) == 3)
        Rotate(s4_r1, s4_r2);
    if ((i % 8) == 4)
        Rotate(s5_r1, s5_r2);
    if ((i % 8) == 5)
        Rotate(s6_r1, s6_r2);
    if ((i % 8) == 6)
        Rotate(s7_r1, s7_r2);
    if ((i % 8) == 7)
        Rotate(s8_r1, s8_r2);

    // Simulate mixing layer and add constraints
    error = MixColumns(model);
    if (error)
        return NULL;
}

// excluding the trivial solution with no active sboxes at all
for (i=0; i < ROWS * COLUMNS * ROUNDS; i++)
{
    varindex[i] = i;
    varcoeff[i] = 1;
}
// add the constraint
error = GRBaddconstr(model, ROUNDS*ROWS*COLUMNS, varindex, varcoeff, GRB_GREATER_EQUAL, 1.0,
     NULL);
```

```
    if (error)
        return NULL;

    return model;
}

// Solves a MILPmodel and returns the optimized objective value; returns 0 if error occured
int solve_MILPmodel(GRBmodel* model)
{
    double solution;
    int status;

    // solve
    error = GRBoptimize(model);
    if (error)
        return 0;

    // check if model is solved optimal
    error = GRBgetintattr(model, GRB_INT_ATTR_STATUS, &status);
    if (error || (status != GRB_OPTIMAL))
  return 0;

    // get the solution
    error = GRBgetdblattr(model, GRB_DBL_ATTR_OBJVAL, &solution);
    if (error)
        return 0;

    // for integer cast
    solution = solution + 0.3;

    return (int)solution;
}

int main()
{
  // Variable for the solution and iteration counters over all shiftings. All permutations will
       have 0 shifting in first row
  int activeSBoxes;
  int ctr_PI1row2, ctr_PI2row2, ctr_PI3row2, ctr_PI4row2, ctr_PI5row2, ctr_PI6row2, ctr_PI7row2,
       ctr_PI8row2;

  // File to print the solution in
  FILE *fp;
  fp = fopen("solution.txt", "w");

  // Create gurobi environment
  GRBenv *environment = NULL;
  error = GRBloadenv(&environment, NULL);
  if (error || environment == NULL)
    return 1;

  // switch off presolving, if error occurs
  #ifdef NOT_PRESOLVING
    error = GRBsetintparam(environment, "PRESOLVE", 0);
    if (error)
        return 1;
  #endif

  // Create an empty MILPModel which will handle all the minimization problems
  GRBmodel* minimizeActiveSBoxes = NULL;

  #ifdef SINGLE_ROTATIONS
```

```
// Iterate over all possible single rotations and solve the model
fprintf(fp, "single cyclic rotations: \n\n");
      for(ctr_PI1row2=0; ctr_PI1row2<(int)(COLUMNS/2)+1; ctr_PI1row2++)
      {
          ctr_PI2row2 = ctr_PI1row2; ctr_PI3row2 = ctr_PI1row2;
          ctr_PI4row2 = ctr_PI1row2; ctr_PI5row2 = ctr_PI1row2;
          ctr_PI6row2 = ctr_PI1row2; ctr_PI7row2 = ctr_PI1row2;
          ctr_PI8row2 = ctr_PI1row2;
          minimizeActiveSBoxes = create_MILPmodel(environment, 0, ctr_PI1row2, 0, ctr_PI2row2,
              0, ctr_PI3row2, 0, ctr_PI4row2, 0, ctr_PI5row2, 0, ctr_PI6row2,
              0, ctr_PI7row2, 0, ctr_PI8row2);

          activeSBoxes = solve_MILPmodel(minimizeActiveSBoxes);

        // Free the model
          GRBfreemodel(minimizeActiveSBoxes);

          // Print solution in file and check if error occured
          if (!activeSBoxes)
            fprintf(fp, "Error testing shifting!\n");
          else
            fprintf(fp, "guaranteed %i-round trail weight: %i S: %i - %i\n", ROUNDS, activeSBoxes,
                  0, ctr_PI1row2);
      }
#endif

#ifdef TWO_ALTERNATING
// Now iterate over all possible 2-alternating rotations and solve the model
fprintf(fp, "\n\n2-alternating cyclic rotations: \n\n");
      for(ctr_PI1row2=0; ctr_PI1row2<(int)(COLUMNS/2)+1; ctr_PI1row2++)
      {
        for(ctr_PI2row2=0; ctr_PI2row2<(int)(COLUMNS/2)+1; ctr_PI2row2++)
        {
          ctr_PI3row2 = ctr_PI1row2; ctr_PI4row2 = ctr_PI2row2;
          ctr_PI5row2 = ctr_PI1row2; ctr_PI6row2 = ctr_PI2row2;
          ctr_PI7row2 = ctr_PI1row2; ctr_PI8row2 = ctr_PI2row2;
          minimizeActiveSBoxes = create_MILPmodel(environment, 0, ctr_PI1row2, 0, ctr_PI2row2,
              0, ctr_PI3row2, 0, ctr_PI4row2, 0, ctr_PI5row2, 0, ctr_PI6row2,
              0, ctr_PI7row2, 0, ctr_PI8row2);

          activeSBoxes = solve_MILPmodel(minimizeActiveSBoxes);

        // Free the model
          GRBfreemodel(minimizeActiveSBoxes);

          // Print solution in file and check if error occured
          if (!activeSBoxes)
            fprintf(fp, "Error testing shifting!\n");
          else
            fprintf(fp, "guaranteed %i-round trail weight: %i S1: %i - %i S2: %i - %i\n", ROUNDS,
                  activeSBoxes, 0, ctr_PI1row2, 0, ctr_PI2row2);
        }
      }
#endif

#ifdef FOUR_ALTERNATING
// Now iterate over all possible 4-alternating rotations and solve the model
fprintf(fp, "\n\n4-alternating cyclic rotations (only optimum printed): \n\n");
      for(ctr_PI1row2=0; ctr_PI1row2<(int)(COLUMNS/2)+1; ctr_PI1row2++)
      {
        for(ctr_PI2row2=0; ctr_PI2row2<(int)(COLUMNS/2)+1; ctr_PI2row2++)
        {
```

```
        for(ctr_PI3row2=0; ctr_PI3row2<(int)(COLUMNS/2)+1; ctr_PI3row2++)
        {
        for(ctr_PI4row2=0; ctr_PI4row2<(int)(COLUMNS/2)+1; ctr_PI4row2++)
        {
          ctr_PI5row2 = ctr_PI1row2; ctr_PI6row2 = ctr_PI2row2;
          ctr_PI7row2 = ctr_PI3row2; ctr_PI8row2 = ctr_PI4row2;
          minimizeActiveSBoxes = create_MILPmodel(environment, 0, ctr_PI1row2, 0, ctr_PI2row2,
              0, ctr_PI3row2, 0, ctr_PI4row2, 0, ctr_PI5row2, 0, ctr_PI6row2,
              0, ctr_PI7row2, 0, ctr_PI8row2);

          activeSBoxes = solve_MILPmodel(minimizeActiveSBoxes);

        // Free the model
          GRBfreemodel(minimizeActiveSBoxes);

          // Print solution in file and check if error occurred
          if (!activeSBoxes || (activeSBoxes >= BRANCHNMB*BRANCHNMB*BRANCHNMB))
          {
          if (!activeSBoxes)
            fprintf(fp, "Error testing shifting!\n");
          else
            fprintf(fp, "guaranteed %i-round trail weight: %i S1: %i - %i S2: %i - %i S3: %i - %i
                 S4: %i - %i\n",
              ROUNDS, activeSBoxes, 0, ctr_PI1row2, 0, ctr_PI2row2, 0, ctr_PI3row2, 0, ctr_PI4row2
                  );
          }
        }
        }
        }
        }

#endif

     #ifdef EIGHT_ALTERNATING
     // Now iterate over all possible rotations in an 8-round trail and solve the model
      fprintf(fp, "\n\nAll possible cyclic rotations (only optimum printed): \n\n");
      // This is all independent from first and last permutation
      ctr_PI1row2 = 0;
      ctr_PI8row2 = 0;
      for(ctr_PI2row2=0; ctr_PI2row2<(int)(COLUMNS/2)+1; ctr_PI2row2++)
      {
        for(ctr_PI3row2=0; ctr_PI3row2<(int)(COLUMNS/2)+1; ctr_PI3row2++)
        {
        for(ctr_PI4row2=0; ctr_PI4row2<(int)(COLUMNS/2)+1; ctr_PI4row2++)
        {
        for(ctr_PI5row2=0; ctr_PI5row2<(int)(COLUMNS/2)+1; ctr_PI5row2++)
        {
        for(ctr_PI6row2=0; ctr_PI6row2<(int)(COLUMNS/2)+1; ctr_PI6row2++)
        {
        for(ctr_PI7row2=0; ctr_PI7row2<(int)(COLUMNS/2)+1; ctr_PI7row2++)
        {
          minimizeActiveSBoxes = create_MILPmodel(environment, 0, ctr_PI1row2, 0, ctr_PI2row2,
              0, ctr_PI3row2, 0, ctr_PI4row2, 0, ctr_PI5row2, 0, ctr_PI6row2,
              0, ctr_PI7row2, 0, ctr_PI8row2);

          activeSBoxes = solve_MILPmodel(minimizeActiveSBoxes);

        // Free the model
          GRBfreemodel(minimizeActiveSBoxes);

          // Print solution in file and check if error occured
          if (!activeSBoxes || (activeSBoxes >= BRANCHNMB*BRANCHNMB*BRANCHNMB))
```

```
          {
          if (!activeSBoxes)
            fprintf(fp, "Error testing shifting!\n");
          else
            fprintf(fp, "guaranteed %i-round trail weight: %i S1: %i-%i S2: %i-%i S3: %i-%i S4: %
                 i-%i S5: %i-%i S6: %i-%i S7: %i-%i S8: %i-%i\n",
              ROUNDS, activeSBoxes, 0, ctr_PI1row2, 0, ctr_PI2row2, 0, ctr_PI3row2, 0, ctr_PI4row2
                   ,
              0, ctr_PI5row2, 0, ctr_PI6row2, 0, ctr_PI7row2, 0, ctr_PI8row2);
        }
        }
        }
        }
        }
      }
    }
    #endif


  // Free the environment
  GRBfreeenv(environment);

  return 0;
}
```

Listing A.4: Brute force approach on 2-alternating shifting for dimension $2 \times x$

It outputs a list of all results. For the special case of a cubed state, this provides

```
single cyclic rotations:

guaranteed 8-round trail weight: 12 S: 0 - 0
guaranteed 8-round trail weight: 24 S: 0 - 1
guaranteed 8-round trail weight: 18 S: 0 - 2


2-alternating cyclic rotations:

guaranteed 8-round trail weight: 12 S1: 0 - 0  S2: 0 - 0
guaranteed 8-round trail weight: 12 S1: 0 - 0  S2: 0 - 1
guaranteed 8-round trail weight: 12 S1: 0 - 0  S2: 0 - 2
guaranteed 8-round trail weight: 12 S1: 0 - 1  S2: 0 - 0
guaranteed 8-round trail weight: 24 S1: 0 - 1  S2: 0 - 1
guaranteed 8-round trail weight: 24 S1: 0 - 1  S2: 0 - 2
guaranteed 8-round trail weight: 12 S1: 0 - 2  S2: 0 - 0
guaranteed 8-round trail weight: 24 S1: 0 - 2  S2: 0 - 1
guaranteed 8-round trail weight: 18 S1: 0 - 2  S2: 0 - 2
```

Listing A.5: List of results for dimension 2×4

Listing A.6 is a presentation of all 2-alternating arbitrary permutations in dimension 2×4, which guarantee an eight-round trail weight of 27 for an optimal branch number of 3.

```
Guaranteed 8-round trail weight of 27 for 2-alternating arbitrary permutations in 2 x 4:

#  1  P1:0,2,4,6 - 3,1,7,5  P2:2,0,6,4 - 3,1,7,5    #  2  P1:0,2,4,6 - 3,1,7,5  P2:2,0,6,4 - 3,5,7,1
#  3  P1:0,2,4,6 - 3,1,7,5  P2:2,0,6,4 - 5,7,1,3    #  4  P1:0,2,4,6 - 3,1,7,5  P2:2,0,6,4 - 7,1,3,5
#  5  P1:0,2,4,6 - 3,1,7,5  P2:2,4,6,0 - 3,1,7,5    #  6  P1:0,2,4,6 - 3,1,7,5  P2:2,4,6,0 - 3,5,7,1
#  7  P1:0,2,4,6 - 3,1,7,5  P2:2,4,6,0 - 5,7,1,3    #  8  P1:0,2,4,6 - 3,1,7,5  P2:2,4,6,0 - 7,1,3,5
#  9  P1:0,2,4,6 - 3,1,7,5  P2:4,6,0,2 - 3,1,7,5    # 10  P1:0,2,4,6 - 3,1,7,5  P2:4,6,0,2 - 3,5,7,1
# 11  P1:0,2,4,6 - 3,1,7,5  P2:4,6,0,2 - 5,7,1,3    # 12  P1:0,2,4,6 - 3,1,7,5  P2:4,6,0,2 - 7,1,3,5
# 13  P1:0,2,4,6 - 3,1,7,5  P2:6,0,2,4 - 3,1,7,5    # 14  P1:0,2,4,6 - 3,1,7,5  P2:6,0,2,4 - 3,5,7,1
# 15  P1:0,2,4,6 - 3,1,7,5  P2:6,0,2,4 - 5,7,1,3    # 16  P1:0,2,4,6 - 3,1,7,5  P2:6,0,2,4 - 7,1,3,5
# 17  P1:0,2,4,6 - 3,5,7,1  P2:2,0,6,4 - 3,1,7,5    # 18  P1:0,2,4,6 - 3,5,7,1  P2:2,0,6,4 - 3,5,7,1
# 19  P1:0,2,4,6 - 3,5,7,1  P2:2,0,6,4 - 5,7,1,3    # 20  P1:0,2,4,6 - 3,5,7,1  P2:2,0,6,4 - 7,1,3,5
# 21  P1:0,2,4,6 - 3,5,7,1  P2:2,4,6,0 - 3,1,7,5    # 22  P1:0,2,4,6 - 3,5,7,1  P2:2,4,6,0 - 3,5,7,1
# 23  P1:0,2,4,6 - 3,5,7,1  P2:2,4,6,0 - 5,7,1,3    # 24  P1:0,2,4,6 - 3,5,7,1  P2:2,4,6,0 - 7,1,3,5
# 25  P1:0,2,4,6 - 3,5,7,1  P2:4,6,0,2 - 3,1,7,5    # 26  P1:0,2,4,6 - 3,5,7,1  P2:4,6,0,2 - 3,5,7,1
# 27  P1:0,2,4,6 - 3,5,7,1  P2:4,6,0,2 - 5,7,1,3    # 28  P1:0,2,4,6 - 3,5,7,1  P2:4,6,0,2 - 7,1,3,5
# 29  P1:0,2,4,6 - 3,5,7,1  P2:6,0,2,4 - 3,1,7,5    # 30  P1:0,2,4,6 - 3,5,7,1  P2:6,0,2,4 - 3,5,7,1
# 31  P1:0,2,4,6 - 3,5,7,1  P2:6,0,2,4 - 5,7,1,3    # 32  P1:0,2,4,6 - 3,5,7,1  P2:6,0,2,4 - 7,1,3,5
# 33  P1:0,2,4,6 - 5,7,1,3  P2:2,0,6,4 - 3,1,7,5    # 34  P1:0,2,4,6 - 5,7,1,3  P2:2,0,6,4 - 3,5,7,1
# 35  P1:0,2,4,6 - 5,7,1,3  P2:2,0,6,4 - 5,7,1,3    # 36  P1:0,2,4,6 - 5,7,1,3  P2:2,0,6,4 - 7,1,3,5
# 37  P1:0,2,4,6 - 5,7,1,3  P2:2,4,6,0 - 3,1,7,5    # 38  P1:0,2,4,6 - 5,7,1,3  P2:2,4,6,0 - 3,5,7,1
# 39  P1:0,2,4,6 - 5,7,1,3  P2:2,4,6,0 - 5,7,1,3    # 40  P1:0,2,4,6 - 5,7,1,3  P2:2,4,6,0 - 7,1,3,5
# 41  P1:0,2,4,6 - 5,7,1,3  P2:4,6,0,2 - 3,1,7,5    # 42  P1:0,2,4,6 - 5,7,1,3  P2:4,6,0,2 - 3,5,7,1
# 43  P1:0,2,4,6 - 5,7,1,3  P2:4,6,0,2 - 5,7,1,3    # 44  P1:0,2,4,6 - 5,7,1,3  P2:4,6,0,2 - 7,1,3,5
# 45  P1:0,2,4,6 - 5,7,1,3  P2:6,0,2,4 - 3,1,7,5    # 46  P1:0,2,4,6 - 5,7,1,3  P2:6,0,2,4 - 3,5,7,1
# 47  P1:0,2,4,6 - 5,7,1,3  P2:6,0,2,4 - 5,7,1,3    # 48  P1:0,2,4,6 - 5,7,1,3  P2:6,0,2,4 - 7,1,3,5
# 49  P1:0,2,4,6 - 7,1,3,5  P2:2,0,6,4 - 3,1,7,5    # 50  P1:0,2,4,6 - 7,1,3,5  P2:2,0,6,4 - 3,5,7,1
# 51  P1:0,2,4,6 - 7,1,3,5  P2:2,0,6,4 - 5,7,1,3    # 52  P1:0,2,4,6 - 7,1,3,5  P2:2,0,6,4 - 7,1,3,5
# 53  P1:0,2,4,6 - 7,1,3,5  P2:2,4,6,0 - 3,1,7,5    # 54  P1:0,2,4,6 - 7,1,3,5  P2:2,4,6,0 - 3,5,7,1
# 55  P1:0,2,4,6 - 7,1,3,5  P2:2,4,6,0 - 5,7,1,3    # 56  P1:0,2,4,6 - 7,1,3,5  P2:2,4,6,0 - 7,1,3,5
# 57  P1:0,2,4,6 - 7,1,3,5  P2:4,6,0,2 - 3,1,7,5    # 58  P1:0,2,4,6 - 7,1,3,5  P2:4,6,0,2 - 3,5,7,1
# 59  P1:0,2,4,6 - 7,1,3,5  P2:4,6,0,2 - 5,7,1,3    # 60  P1:0,2,4,6 - 7,1,3,5  P2:4,6,0,2 - 7,1,3,5
# 61  P1:0,2,4,6 - 7,1,3,5  P2:6,0,2,4 - 3,1,7,5    # 62  P1:0,2,4,6 - 7,1,3,5  P2:6,0,2,4 - 3,5,7,1
# 63  P1:0,2,4,6 - 7,1,3,5  P2:6,0,2,4 - 5,7,1,3    # 64  P1:0,2,4,6 - 7,1,3,5  P2:6,0,2,4 - 7,1,3,5
```

Listing A.6: List of optimal results for 2-alternating arbitrary permutations in the smallest dimension

List of Figures

2.1 Illustration of the AES state. 4
2.2 Simple interpretation of the state with dimension $2 \times 2 \times 2$. 7
2.3 Illustration of one round r of the AES-encryption. 8

3.1 Pictograms for permutations on seperate rows or columns. 14

4.1 Illustration of the permutations 0-1 and $\mathtt{SR}$ for dimension 2×4. 26
4.2 Illustration of the cube rotations rot(90°) and rot(120°). 26
4.3 Optimal four-round trail in the AES using the standard `ShiftRows` layer. 29

5.1 Position of the 4 megaboxes in a 4×16 state. 37
5.2 Mixing of megaboxes after applying $\mathtt{P}_{\sigma_1}$. 38

List of Tables

2.1 Number of rounds for certain key length [PP10]. 4

4.1 Mimimum number of active bundles in a 2×4 state for a few examples. . 27
4.2 Mimimum number of active bundles in a 3×9 state for a few examples. . 28
4.3 Mimimum number of active bundles in a 4×16 state for a few examples. 28
4.4 Mimimum number of active bundles in a 4×16 state for two-alternating shiftings. 32
4.5 Optimal bound in a 4×16 state with four-alternating shiftings. 33
4.6 Optimal non-alternating shiftings in a 2×4 state. 34

Bibliography

[ADH98] A.S. Asratian, T.M.J. Denley, and R. Häggkvist. *Bipartite Graphs and Their Applications.* Cambridge Tracts in Mathematics. Cambridge University Press, 1998.

[BBG+09] R. Benadjila, O. Billet, H. Gilbert, G. Macario-Rat, T. Peyrin, M. Robshaw, and Y. Seurin. Sha-3 proposal: Echo. *Submission to NIST (updated)*, pages 1–5, 2009.

[BDPVA11] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. On alignment in keccak. In *ECRYPT II Hash Workshop*, volume 51, page 122, 2011.

[BK14] A. Biryukov and D. Khovratovich. Paeq v1. 2014. Available online at `http://competitions.cr.yp.to/round1/paeqv1.pdf`.

[BS91] E. Biham and A. Shamir. Differential cryptanalysis of des-like cryptosystems. *Journal of CRYPTOLOGY*, 4(1):3–72, 1991.

[DR98] J. Daemen and V. Rijmen. Aes proposal: Rijndael, 1998. Updated version available online at `http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf`, 2003.

[DR02] J. Daemen and V. Rijmen. *The Design of Rijndael.* Springer–Verlag, Berlin Heidelberg, 2002.

[DR06] J. Daemen and V. Rijmen. Understanding two-round differentials in aes. In *Security and Cryptography for Networks*, pages 78–94. Springer, 2006.

[FIP01] PUB FIPS. 197: Advanced encryption standard (aes). *National Institute of Standards and Technology*, 2001. Available online at `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`.

[GKM+08] P. Gauravaram, L. R. Knudsen, K. Matusiewicz, F. Mendel, C. Rechberger, M. Schläffer, and S. Thomsen. Grøstl–a sha-3 candidate. *Submission to NIST*, 2008. Available online at `http://www.groestl.info/Groestl-0.pdf`.

[go] gurobi optimization. Documentation and examples. `http://www.gurobi.com/resources/documentation`. Last seen on 8th July 2014.

[GP10] H. Gilbert and T. Peyrin. Super-sbox cryptanalysis: improved attacks for aes-like permutations. In *Fast Software Encryption*, pages 365–383. Springer, 2010.

[GPPR11] J. Guo, T. Peyrin, A. Poschmann, and M. Robshaw. The led block cipher. In *Cryptographic Hardware and Embedded Systems–CHES 2011*, pages 326–341. Springer, 2011.

[Kal02] J. Kallrath. *Gemischt-ganzzahlige Optimierung: Modellierung in der Praxis*. Vieweg Verlag, 2002.

[KR11] L. R. Knudsen and M. J. B. Robshaw. *The Block Cipher Companion*. Springer–Verlag, Berlin Heidelberg, 2011.

[MS78] F.J. MacWilliams and N.J.A. Sloane. *The Theory of Error-Correcting Codes*. North-holland Publishing Company, 2nd edition, 1978.

[MT12] B. Meindl and M. Templ. Analysis of commercial and free and open source solvers for linear optimization problems. *Eurostat and Statistics Netherlands within the project ESSnet on common tools and harmonised methodology for SDC in the ESS*, 2012.

[MWGP11] N. Mouha, Q. Wang, D. Gu, and B. Preneel. Differential and linear cryptanalysis using mixed-integer linear programming. *Lecture Notes of Computer Science*, 7537:57–76, 2011.

[PP10] C. Paar and J. Pelzl. *Understanding Cryptography*. Springer–Verlag, Berlin Heidelberg, 2010.

[RDP+96] V. Rijmen, J. Daemen, B. Preneel, A. Bosselaers, and E. De Win. The cipher shark. In *Fast Software Encryption*, pages 99–111. Springer, 1996.

[sag] sagemath. Mixed integer linear programming. `http://www.sagemath.org/doc/reference/numerical/sage/numerical/mip.html`. Last seen on 10th April 2014.

[Sha49] C. E. Shannon. Communication theory of secrecy systems*. *Bell system technical journal*, 28(4):656–715, 1949.

[Wel88] D. Welsh. *Codes and cryptography*. Clarendon Press, 1988.