Bachelor's Thesis Computing Science

Radboud University Nijmegen

An Algorithmic Approach to Fixing Friet

Author:
Kutay Sezer
s1075213

First supervisors/assessors:
Professor Joan Daemen,
PhD-candidate Jan Schoone

Second assessor:
Associate Professor Bart Mennink

August 18, 2024

Abstract

In this paper, we address the vulnerabilities that were discovered in the authenticated encryption scheme Friet [14]. The authors of [17] demonstrated that the scheme is susceptible to common cryptographic attacks that exploit input-output relations. To mitigate these exploitable weaknesses, we propose a methodology that is aimed at redesigning the linear component of the Friet primitive -the underlying function of Friet- to improve its security. Our approach involves the analysis of matrix-based permutations. We begin with square matrices with small dimensions and gradually work our way up to analyze longer patterns. We prioritize ensuring that any potential redesign is resistant to differential and linear cryptanalysis, which are powerful techniques for attacking cryptographic schemes. Additionally, we consider the algebraic degree of a potential redesign, which is another useful metric for security. Throughout this paper, we conduct a series of analyses of the relationship between structural properties of matrices and their impact on the primitive's resistance to attacks. Although we did not manage to find a fix for the Friet primitive, we did manage to highlight unique correlations between matrix properties and cryptographic security. Our research as a result, significantly narrows the search space required for future investigations of matrices and offers a methodology for constructing a permutation that follows specified criteria.

Contents

1 Introduction **3**
- 1.1 Problem Statement . . . 3
- 1.2 Methodology . . . 3

2 Preliminaries **5**
- 2.1 Notation and Basic Definitions . . . 5
  - 2.1.1 Finite Field Arithmetic . . . 5
  - 2.1.2 Toffoli Gate . . . 5
  - 2.1.3 Linear Algebra . . . 6
  - 2.1.4 Bitstrings . . . 9
- 2.2 Cryptography Basics . . . 10
  - 2.2.1 Encryption Scheme . . . 11
  - 2.2.2 Permutations . . . 11
  - 2.2.3 Round Function . . . 11
  - 2.2.4 Distinguishers . . . 11
  - 2.2.5 Advantage . . . 11
  - 2.2.6 Security Claim . . . 12
  - 2.2.7 Security Strength . . . 12
  - 2.2.8 PRP Security Notion . . . 12
  - 2.2.9 The Even–Mansour Construction . . . 13
- 2.3 Cryptanalysis Techniques . . . 14
  - 2.3.1 Differential Cryptanalysis . . . 14
  - 2.3.2 Linear Cryptanalysis . . . 18
  - 2.3.3 Algebraic Degree . . . 20
- 2.4 Table of Definitions . . . 21

3 Friet Definition and Concepts **24**

4 The Weaknesses of Friet-PC **26**

5 Strategy of Fixing Friet-PC **29**

6 Unsuccessful Attempts at Fixing the Primitive **30**
6.1 Attempt 1 30
6.2 Attempt 2 31
6.3 Attempt 3 32
6.4 Testing the Quality of the Candidates 33
6.5 The Design Flaw of the Initial Attempts 38

7 Investigation of Suitable Linear Transformations **41**
7.1 What Makes a Matrix Suitable 42

8 Measuring the Security of the Permutation with a Chosen Matrix **50**

9 Algebraic Degree Analysis **60**

10 Conclusions **67**

11 Future Work **69**

Chapter 1

Introduction

In 2020 EUROCRYPT, a paper introducing a new authenticated encryption scheme with built-in fault detection mechanisms called Friet was published.[14] Friet is a permutation-based encryption scheme, its permutation Friet-P is constructed with a smaller permutation Friet-PC. The authors report on the performance in software and hardware of the permutation used in Friet. They additionally evaluate the fault-detection capabilities of the software and simulate hardware implementations with attacks. The authors of Friet claimed a security strength of 128 bits for the confidentiality and integrity of the Friet encryption scheme.

However, in 2022, an article was published that outlined an attack methodology breaking the confidentiality and integrity claims for Friet.[17] It highlights overlooked weaknesses against differential and linear cryptanalysis. In the article, the authors prove the existence of properties that, over n rounds, can be exploited with a probability of 1. They showcase that the weakness of Friet underlies in the design of the primitive Friet-PC.

1.1 Problem Statement

In this paper, we aim to introduce a ground-up methodology for fixing Friet and discuss the properties a potential fix would need to have.

Additionally, we aimed to highlight interesting properties we found regarding matrices concerning cryptanalysis and algebraic degrees.

1.2 Methodology

We first try to fix the primitive by redesigning the linear section of the round function while conserving the design rationale of the primitive and testing these redesigns. After discovering that none of the redesigns proved to be secure against linear cryptanalysis. We adopt a new strategy. Instead of making small modifications to the linear part of Friet-PC, we decided on a different approach. We investigated whether any possible fix to

the linear section could make the new primitive resistant to both differential and linear cryptanalysis regardless of keeping the step functions defined in Friet-PC (3.1). Our goal was to find a solution that would improve the security of the system against these types of cryptanalysis. We interpreted the linear section as a matrix of size 384×384, however, due to the size of the search space required, we took advantage of the properties of repeating patterns of bits. This allowed us to start from 3×3 matrices, expand them to 12×12, and analyze permutations constructed from them. In order, we tested for "trail analysis for potentially dangerous input-output relations", "lowest weight trail analysis" and "algebraic degree analysis".

Chapter 2

Preliminaries

In this section, we discuss core concepts in Finite Field Theory, important matrix concepts, measuring the security strength of block ciphers, cryptanalysis concepts & terms, and what an algebraic degree is.

2.1 Notation and Basic Definitions

Here, we give definitions to the fundamental mathematical concepts used in this paper.

2.1.1 Finite Field Arithmetic

A finite field with n elements $\mathbb{F}_n$ is a set with the operations multiplication and addition where the operations satisfy certain rules.

More formally, a set $\mathcal{F}$ with the operations $+$ and $\cdot$ with the identity elements 0 and 1 where $0 \neq 1$ form a field if:

The set of all elements of $\mathcal{F}$ and operation $+$ form an abelian group.
The set of all non-zero elements of $\mathcal{F}$ and operation $\cdot$ form an abelian group.
$a(b+c) = ab + ac$ and $(a+b)c = ac + bc$ for all $a, b, c \in \mathcal{F}$

[1]

This paper only focuses on the finite field with two elements $\mathbb{F}_2 = \{0, 1\}$. In $\mathbb{F}_2$, multiplication and addition operations are done in mod 2.
The XOR operation ($\oplus$) and bitwise AND operation ($\wedge$) are synonymous with addition and multiplication operations over $\mathbb{F}_2$. In this paper, we refer to elements of $\mathbb{F}_2$ as bits.

2.1.2 Toffoli Gate

A Toffoli gate maps bits (a, b, c) to $(a, b, (c+ab))$, which we denote as $(a, b, c) \rightarrow (a, b, c \oplus a \wedge b)$ [12, 29]
The Toffoli gate is applied to individual bits. However, this operation can be extended to bitstrings by applying the Toffoli gate in parallel to each bit. This way, we can define

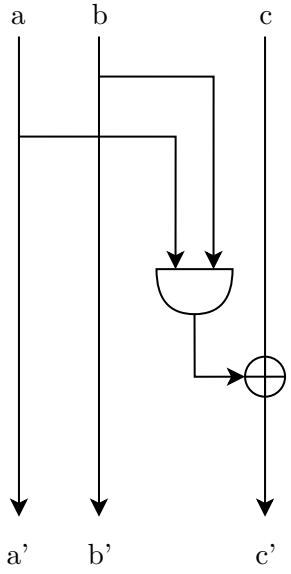


Figure 2.1: Visual representation of the Toffoli gate

an operation that takes three bitstrings of length n and apply the Toffoli gate on each bit. We will refer to this construct as the *Toffoli Stream operation.*

2.1.3 Linear Algebra

In this paper, we regularly make use of linear algebra concepts. In the following sections, we highlight useful matrix concepts and field concepts.

Field Arithmetic with Matrices

Similar to what we discussed in Section 2.1.1, we can perform arithmetic operations with matrices over finite fields as well. A **general linear group** of degree n is the set of $n \times n$ invertible matrices, together with the operation of matrix multiplication. Like matrix multiplication over real numbers, the matrix multiplication over finite fields is not commutative. [10] We are going to focus on the general linear groups of degree n over the finite field $\mathbb{F}_2$, which is denoted as $GL(n, 2)$.

Below is an example of matrix multiplication over the finite field $\mathbb{F}_2$.

Example:

$$\begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} =$$

$$\begin{bmatrix} 1\cdot 0+0\cdot 1+1\cdot 0 & 1\cdot 0+0\cdot 1+1\cdot 1 & 1\cdot 1+0\cdot 1+1\cdot 0 \\ 0\cdot 0+0\cdot 1+1\cdot 0 & 0\cdot 0+0\cdot 1+1\cdot 1 & 0\cdot 1+0\cdot 1+1\cdot 0 \\ 1\cdot 0+1\cdot 1+0\cdot 0 & 1\cdot 0+1\cdot 1+0\cdot 1 & 1\cdot 1+1\cdot 1+0\cdot 0 \end{bmatrix}$$

$$= \begin{bmatrix} 0 & 1 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix}$$

The three matrices above are a part of $GL(3, 2)$.

Invertibility Criterion

A matrix is invertible if and only if its determinant denoted with $det(M)$ for matrix M, (modulo 2 if over $\mathbb{F}_2$) is not equal to 0. For example, all three matrices we discussed in the previous section have a determinant of 1.

Order of a General Linear Group

The order of a general linear group is the size of the group. Over finite fields with q elements, the order of a general linear group of degree n is given as:

$$\prod_{k=0}^{n-1}(q^n - q^k) \quad [10]$$

Example: The number of invertible 4×4 matrices over the finite field $\mathbb{F}_2$, i.e, the order of the general linear group of degree 4 over the finite field with 2 elements is:

$$\prod_{k=0}^{3}(2^4 - 2^k) = (16 - 2^0)(16 - 2^1)(16 - 2^2)(16 - 2^3) = 15 \cdot 14 \cdot 12 \cdot 8 = 20160$$

The order of a matrix M from a general linear group over a finite field is the smallest natural number $q > 0$ such that $M^q = I$. The order q divides the order of the general linear group. This fact can be derived from **Lagrange's Theorem**. [6]

Circulant Matrices

An $n \times n$ circulant matrix C takes the form

$$C = \begin{bmatrix} c_0 & c_1 & \dots & c_{n-2} & c_{n-1} \\ c_{n-1} & c_0 & c_1 & & c_{n-2} \\ \vdots & c_{n-1} & c_0 & \ddots & \vdots \\ c_2 & & \ddots & \ddots & c_1 \\ c_1 & c_2 & \dots & c_{n-1} & c_0 \end{bmatrix}$$

A circulant matrix C is specified by one vector, $c = (c_0, c_1, c_2, \dots, c_{n-2}, c_{n-1})$. Matrix C is constructed by cyclically permutating the entries of values c_i of the vector c (with

offset equal to the row index) [9]

Example: Take the vector $c = (1, 0, 1, 1)$. Then the circulant matrix would look as follows:

$$C = \begin{bmatrix} c_0 & c_1 & c_2 & c_3 \\ c_3 & c_0 & c_1 & c_2 \\ c_2 & c_3 & c_0 & c_1 \\ c_1 & c_2 & c_3 & c_0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \end{bmatrix}$$

Circulant matrices with the matrix multiplication operation form a group. The invertibility of a circulant matrix depends on the vector that forms it. It is mentioned in [8] that if a row has an even number of ones, i.e., if the Hamming weight of the vector forming the circulant matrix is even, then the circulant matrix is singular (not invertible). This means half the circulant matrices that can be formed with vector $c \in \mathbb{F}_2^n$ are not invertible. On the contrary, the other half is invertible. Therefore, the order of the group of $n \times n$ invertible circulant matrices over $\mathbb{F}_2$ is 2^{n-1}.

Block Matrices

Block matrices are matrices in the form of:

$$\begin{bmatrix} A_{0,0} & A_{0,1} & \dots & A_{0,m} \\ A_{1,0} & A_{1,1} & \dots & A_{1,m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n,0} & A_{n,1} & \dots & A_{n,m} \end{bmatrix}$$

where $A_{i,j}$ is another matrix.

A block matrix or a partitioned matrix is a matrix that is subdivided into rectangular blocks of elements called blocks or submatrices. [7, 37]

This is a useful notation to represent one giant matrix as a matrix made out of smaller blocks.

Example: Take the circulant matrix example we saw in the previous section. It can be represented as follows:

$$C = \begin{bmatrix} A_{0,0} & A_{0,1} \\ A_{1,0} & A_{1,1} \end{bmatrix}$$

where $A_{0,0} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}, A_{0,1} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}, A_{1,0} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$ and $A_{1,1} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$

The matrices we consider in this paper are block matrices where the blocks are circulant and these, along with the matrix multiplication operation form a group.

2.1.4 Bitstrings

A bitstring is a sequence that consists of only 1s and 0s. A bitstring is an element of $\mathbb{F}_2^*$, the space of all bitstrings. We represent them in a bold text format $\mathbf{1010}\ldots$. A bit vector is a vector over $\mathbb{F}_2$. We represent them in a vector notation $(1, 0, 1, 0, \ldots)$. A bit vector with dimension Bitstrings and bit vectors are very similar concepts but they are not the same thing. However, a bijective mapping from a bitstring $\{1, 0\}^n$ to $\mathbb{F}_2^n$ is possible. This is why we mostly use the bitstring notation to compactly represent bit vectors in this paper. A bitstring x of length n, denoted as $\text{len}(x)$ is an element of $\mathbb{F}_2^n$. A binary operation $(*)$ over $\mathbb{F}_2$ between two bitstrings $x = x_0x_1x_2\ldots$ and $y = y_0y_1y_2\ldots$ is defined as $x * y = z$ where $z_i = x_i * y_i$.

Example:

$$\mathbf{1011} \oplus \mathbf{0110} = (1 \oplus 0)(0 \oplus 1)(1 \oplus 1)(1 \oplus 0) = \mathbf{1101}$$

Circular Shift Operation on Bitstrings

A circular shift operation to some direction is shifting bits in a bitstring to that direction and wrapping the bits that exceed bitstring from the other side.

Example:

$$\underline{\mathbf{001}}\mathbf{011} \lll 4 = \mathbf{11}\underline{\mathbf{0010}}$$

The underlined section of the bitstring is to denote the chunk that wraps around from the right side.

If a bitstring of length n is shifted to the left by k bits where $k \geq n$, then this shifting operation is equal to shifting the bitstring to the left by $k \mod n$.

Suppose we rotate by $k = c \cdot \text{len}(x) + r$ the bitstring x. Where $c \in \mathbb{N}$ and $0 \leq r < \text{len}(x)$. Rotating a bit at position i by $\text{len}(x)$ results in this bit moving i steps to the left, then moving $\text{len}(x) - i$ step to the left from the rightmost position because of wrapping around. However, since the string is $\text{len}(x)$ bits long, a bit that is $\text{len}(x) - i$ from the right side is at the ith position from the left side. Thus, the rotation operation of length $c \cdot \text{len}(x)$ does not result in displacement of a bit from its initial position. So the total rotation is equivalent to rotating by r bits to the left. This is equivalent to rotating by $k \mod \text{len}(x)$ bits.

$$x \lll k = x \lll (k \mod \text{len}(x))$$

Example:

$$\mathbf{1001} \lll 5 = \mathbf{0011} = \mathbf{1001} \lll (5 \mod \text{len}(\mathbf{1001})) = \mathbf{1001} \lll (5 \mod 4) = \mathbf{1001} \lll 1$$

Repeating Bitstring Patterns

We can create a bitstring that is made out of a repeating smaller bitstring pattern.

Example:

$$(\mathbf{10})^4 = \mathbf{10}\|\mathbf{10}\|\mathbf{10}\|\mathbf{10} = \mathbf{10101010}$$

A bitstring made entirely out of a repeating bitstring pattern has useful properties we can use.

Suppose we have the bitstrings $x = p_1^n$, $y = p_2^n$ and $z = p_3^m$ where $\text{len}(p_1) = \text{len}(p_2)$

For unary operations $\delta \in \{f, g_k, h_k\}$ where $f(x) = \bar{x}$, $g_k(x) = x \lll k$, $h_k(x) = x \ggg k$ and $k \in \mathbb{N}$ on bitstring z, it is the case that:

$$\delta(z) = (\delta(p_3))^m$$

Example:

$$(\mathbf{10})^4 \ggg 5 = \mathbf{10101010} \ggg 5 = (\mathbf{10} \ggg 5)^4 = (\mathbf{10} \ggg 1)^4 = (\mathbf{01})^4 = \mathbf{01010101}$$

For binary operations $(*) \in \{\vee, \wedge, \oplus\}$ on x and y, it is the case that:

$$x * y = (p_1 * p_2)^n$$

Example:

$$(\mathbf{10})^2 \oplus (\mathbf{01})^2 = \mathbf{1010} \oplus \mathbf{0101} = (\mathbf{10} \oplus \mathbf{01})^2 = (\mathbf{11})^2 = \mathbf{1111}$$

These properties allow us to reduce the problem of computing the result of a function on an n-bit bitstring (pair) to computing the result of a function on an k-bit bitstring where k is the length of the bitstring pattern that makes up the larger bitstring. I.e., instead of computing function on $f : \mathbb{F}_2^n \to \mathbb{F}_2^n$ directly, we compute it on $f : \mathbb{F}_2^k \to \mathbb{F}_2^k$ on the pattern p that makes up the whole bitstring. This results in the new pattern p'. Then we compute $(p')^{\frac{n}{k}}$ for the final result.

This reduction becomes especially useful for significantly reducing the workload and space needed for doing computations with matrices on repeating patterns. If we have a matrix M and a vector $\vec{v}$, where $\vec{v}$ consists of a single repeating pattern, we can simplify the computation by reducing the dimensions of both M and $\vec{v}$. This is possible because the repeating pattern in $\vec{v}$ implies that many of the operations in the matrix-vector multiplication are redundant. This allows us to compress $\vec{v}$ to just the unique pattern and adjust the matrix M accordingly. For example, suppose we have a vector $\vec{v}$ with dimensions 8×1 that is made up of a repeating pattern of length 2 and a 8×8 matrix M. This vector can be reduced to 2×1 and the matrix can then be reduced to 2×2.

2.2 Cryptography Basics

In this section, we introduce basic cryptography concepts regarding permutations, encryption, and security strength.

2.2.1 Encryption Scheme

An encryption scheme consists of encryption and decryption operations.

Encryption is the cryptographic operation that enables data confidentiality. Data confidentiality is the assurance that only authorized entities get access to the data. [6] Decryption is the operation that recovers the data from the encrypted data. In symmetric cryptography, both the encryption and decryption operations require a secret key. The data that is used in encryption is called a **plaintext** and it is encrypted in a way that only the parties with a secret key can decrypt it back. The encrypted data is called a **ciphertext**.

An encryption scheme can be built in different ways. The most common method is to combine existing building blocks and construct them from the bottom up. We call these building blocks **primitives**. Mostly, the security of the construction depends on the underlying primitive.

2.2.2 Permutations

A permutation of width b is a bijective mapping $\{1,0\}^b \rightarrow \{1,0\}^b$.

A random permutation ($\mathcal{RP}$), sometimes called a randomly chosen permutation, is a function that is uniformly and randomly chosen from the set of all b-bit permutations. $\mathcal{RP}$ is used as a framework for how permutation-based encryption should behave. [6]

A **block cipher** with a block size of b is a b-bit permutation that is parameterized by a secret key. [6] Some of the operations during the permutation process depend on this secret key.

2.2.3 Round Function

A round function is a function used within each iteration of a larger cryptographic algorithm. The number of times function f is applied on some initial input is called the number of rounds. A round function is accompanied by additional round-specific parameters. For example in the context of this paper, a round constant.

2.2.4 Distinguishers

Given the real-world scheme where the queries are made to a block cipher B_K and ideal-world scheme where the queries are made to $\mathcal{RP}$, a **distinguisher** (Adversary) is an algorithm that predicts which one of the two options it is querying. It is denoted with the symbol $\mathcal{D}$.

2.2.5 Advantage

The advantage of a distinguisher $\mathcal{D}$ distinguishing between a block cipher B_K and $\mathcal{RP}$ given the distinguisher $\mathcal{D}$ is as follows:

$$Adv_{\mathcal{D}} = |\Pr(\mathcal{D} = 1 \mid B_K) - \Pr(\mathcal{D} = 1 \mid \mathcal{RP})| \qquad [6]$$

The distinguisher outputs a 1 if it concludes that it is querying P. It outputs 0 otherwise. $\Pr(\mathcal{D} = 1 \mid B_K)$ is the probability of the distinguisher outputting 1, given that it is querying B_K. This is the probability of the distinguisher guessing correctly. $\Pr(\mathcal{D} = 1 \mid \mathcal{I})$ is the probability of the distinguisher outputting 1, given that it is querying $\mathcal{RP}$. This is the probability of the distinguisher guessing incorrectly. If the advantage is $\gg 0$, then the distinguisher performs better than random guessing. If the advantage is ≈ 1, then the distinguisher is close to guessing incorrectly 100% of the time, or it is close to guessing correctly 100% of the time. In both cases, the distinguisher is significantly more effective than random guessing. [6]

2.2.6 Security Claim

A security claim is an unambiguous statement that defines the minimum success probability an attack must have to be considered as breaking the primitive. Breaking is well-defined and usually, it means distinguishing the output of the primitive from a sequence of random bits. The minimum success probability is typically expressed in terms of the attacker's effort in terms of computation (offline complexity) and data obtained from the keyed primitive (online complexity). [6]

A security claim serves as a security specification. A user can assume there are no attacks with higher success probability than specified in the claim. It also serves as a challenge to break the primitive. [6]

With these concepts in mind, we provide an example of what a security claim would look like.

Example: Consider AES with key size k. For any adversary $\mathcal{D}$ with computational complexity N

$$Adv_{\mathcal{D}} \leq \frac{N}{2^k}$$

[6]

2.2.7 Security Strength

The security strength s of a block cipher B_K given the advantage $Adv_{\mathcal{D}}$ for the distinguisher $\mathcal{D}$, can be calculated as follows:

$$\log_2 \left(\frac{M + N}{Adv_{\mathcal{D}}} \right) < s$$

where M is the online complexity and N is the offline complexity. [6]

2.2.8 PRP Security Notion

Suppose we are given an implementation of a block cipher B_K with secret key K and $\mathcal{RP}$.

Suppose an algorithm chose one of the two without revealing which one with equal probability. Assuming we do not know which device we are communicating with, the **Pseudorandom Permutation (PRP) security** notion of security is infeasibility of a distinguisher to distinguish between B_K and $\mathcal{RP}$. We consider an adversary being able to distinguish between the two if the probability of successfully guessing is significantly more than 0.5. We can construct a security claim through online and offline queries, and with how difficult it is to distinguish between B_K and $\mathcal{RP}$.

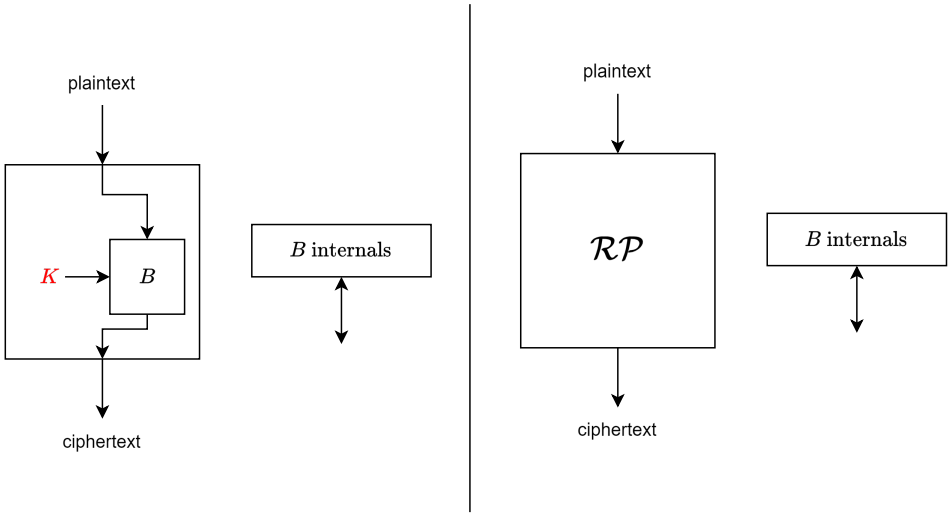


Figure 2.2: Distinguishing attack setup

Offline queries are queries that we make to the internal implementation of B with arbitrary keys. In this case, we have access to the implementation of B so we can implement the algorithm and choose a key K and make queries to B_K.
Online queries are queries that we make to the real instance of B_K or $\mathcal{RP}$, we are not aware of which one we are querying and we do not know the secret key K. [6]

2.2.9 The Even–Mansour Construction

When we have a permutation that we want to measure the security of, a way we can achieve this is by constructing a simple block cipher from it.

A simple method of achieving this is the Even-Mansour construction. First, an XOR operation is applied on the plaintext with key K_1. It is then followed by an application of a publicly known unkeyed permutation F. Finally, an XOR operation is applied to the result with key K_2. This construction allows us to make distinguishing queries between $\mathcal{RP}$ and a block cipher constructed from a permutation F. [4] In the context of this paper, we can show why the properties discovered in the primitive of Friet form a problem.

According to [4], the upper-bound for a block cipher constructed with an Even-Mansour construction would be secure up to $2^{-\frac{b}{2}}$ computational effort where b is the block size.

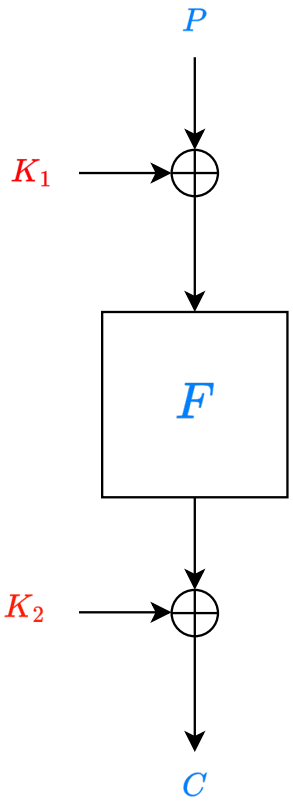


Figure 2.3: Visual representation of the Even-Mansour construction

2.3 Cryptanalysis Techniques

Cryptanalysis of a cryptographic scheme is the study and application of various techniques to break/weaken the scheme by identifying and exploiting its vulnerabilities. There are commonly used methods of cryptanalysis. Three of these are the core topics of this paper. These are called differential cryptanalysis, linear cryptanalysis, and algebraic degree analysis. In the following sections, we define these and their concepts.

2.3.1 Differential Cryptanalysis

Differential cryptanalysis is a type of cryptanalysis that studies how differences in input can affect the resulting difference in the output. We denote a **difference** with either Δ or other Greek letters like α, β, etc. There are different ways a difference is calculated but usually, the XOR operation is used. The difference Δ between a pair of bitstrings is equal to $x \oplus y$.

Given the permutation f, input x and the input difference Δ_{in}, the output difference

Δ_{out} is given as follows:

$$\Delta_{out} = f(x \oplus \Delta_{in}) \oplus f(x)$$

[2]

Differentials and Differential Probability

A **differential** is a pair of differences. It is denoted as (Δ_x, Δ_y) for some difference Δ_x and Δ_y.
The **differential probability**, denoted with $\mathrm{DP}(\Delta_x, \Delta_y)$ is the probability of the output difference being Δ_y, given that the input difference was Δ_x. The differential probability is calculated as such:

$$\mathrm{DP}(\Delta_{in}, \Delta_{out}) = \frac{|\{x \mid x \in \mathbb{F}_2^n,\ f(x) \oplus f(x \oplus \Delta_{in}) = \Delta_{out}\}|}{2^n} \qquad [2]$$

where f is a function and n is the length of the inputs.

The all-zero differential is trivial, since if there is no difference between inputs, then there will be no difference in the output. For this reason, assume that we do not consider the all-zero difference in any of the analyses we did in this paper. [17]

Differential Trail

A differential trail of r-rounds, denoted with $T = (\Delta_0, \Delta_1, \ldots, \Delta_r)$ is the ordered tuple of the differences that we compute with the initial difference Δ_0 and r rounds of round function application on the initial input. The DP of a trail over r rounds is calculated as $DP(T)$ the probability that input pair $(x, x + \Delta_0)$ with x uniformly random will exhibit the sequence of differences through the rounds. [14] Calculating the DP of a trail directly is difficult since the outcome of one round's differential could influence the outcome of another round's differential; instead, we approximate the differential probability of the trail T as follows:

$$\mathrm{DP}(T) \approx \prod_{k=0}^{r-1} \mathrm{DP}_{f_k}(\Delta_k, \Delta_{k+1})$$

where f_k is a round function. This equation approximates the DP of a trail. Assuming differentials in the trail are independent, the equation calculates the exact DP of the trail. The weight of a trail is defined as $-\log_2(\mathrm{DP}(T))$ for a trail T.

Propagation of Differences through Linear Operations

In this section, we list the effects of linear operations on difference(s). Let λ be a linear operation and Δ_{in} the input difference for λ, then Δ_{out} is the image of Δ_{in} resulting from the application of λ. For the differential $(\Delta_{in}, \Delta_{out})$ it is always the case that $\mathrm{DP}(\Delta_{in}, \Delta_{out}) = 1$

Branching Property:

The branching operation is a process where a single input value is duplicated and propagated along multiple outgoing paths. The input is copied to each outgoing arrow and they carry the same input value.

Given the input difference Δ_{in}, the resulting differences branching out of an intersection point will also have the same value.

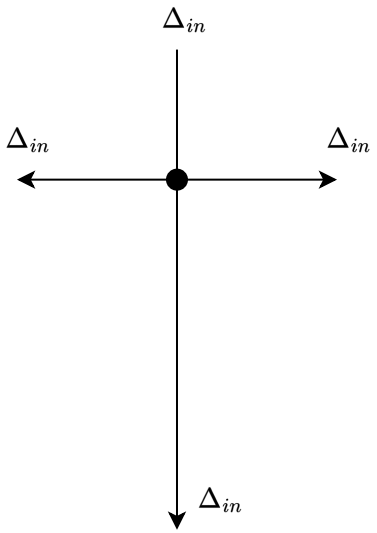


Figure 2.4: Visual representation of the effect of branching operation on differences

Constant XOR Property:

Given an input difference Δ_{in} and a constant value c, the result of the XOR operation will be equal to the input difference. Adding a constant to two inputs that differ by Δ will still differ by the same value since the bits that are changed in one input also change the same way in the other.

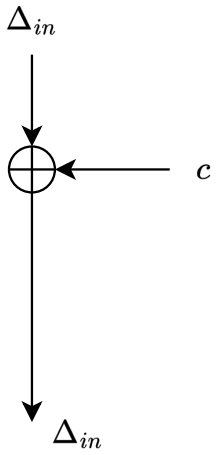


Figure 2.5: Visual representation of the effect of an XOR operation with a constant on differences

XOR Property:

Given incoming differences $\Delta_0, \Delta_1, \ldots, \Delta_n$ the result will be equal to $\bigoplus_{i=0}^{n} \Delta_i$.

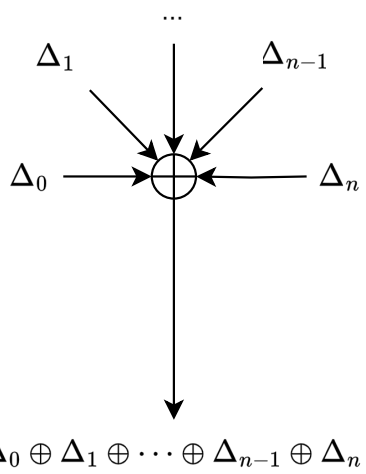


Figure 2.6: Visual representation of the effect of XOR operation differences

Rotation Property:

The rotation operation has the same effect on a difference as it does in a regular input. so a rotation by k bits to the left on the input difference Δ_{in} results in $\Delta_{in} \lll k$.

Generalized Property:

Suppose we have a linear transformation f that modifies an input bitstring and outputs another bitstring. Given the matrix representation M of this linear transformation and the input difference Δ_{in}, the relation between the input difference and output difference Δ_{out} for f is as follows:

$$\Delta_{out} = M \cdot \Delta_{in}$$

This follows from the fact that

$$\text{DP}(\Delta_{in}, \Delta_{out}) = 1 \text{ if and only if } \Delta_{out} = M \cdot \Delta_{in} \text{ and 0 otherwise}$$

2.3.2 Linear Cryptanalysis

Linear cryptanalysis is a method of identifying and representing linear relationships between the plaintext and ciphertext (and key bits).

Linear Masks

Identifying linear relationships between plaintext and ciphertext pairs for a given function is carried out with a pair of bitstrings called **linear masks**, or masks for short. They are denoted with Γ or other Greek letters. A mask is a vector of bits used to denote which bits of a bitstring x are used for the approximation of a function. Given a mask α and a bitstring x, the mask value is calculated as $\alpha \cdot x$ where $(\cdot)$ is the dot product.

Correlation and Linear Potential

A pair of input and output masks is called a **linear approximation**. Correlation measures the "closeness" of the linear approximation of a function f to the function itself. It takes values between -1 and 1. The equation for the correlation of a linear approximation is as follows:

$$\text{Corr}(\Gamma_{in}, \Gamma_{out}) = 2 \cdot \frac{|\{x \mid x \in \mathbb{F}_2^n,\ \Gamma_{in} \cdot x = \Gamma_{out} \cdot f(x)\}|}{2^n} - 1 \qquad [11]$$

where n is the length of the input, f is the original function we are trying to approximate, Γ_{in} and Γ_{out} are the input-output masks and $(\cdot)$ operation is the dot product. Similar to the advantage score in Section 2.2.5, we want the correlation to be far from 1 and -1. I.e., we want the absolute value of the correlation ≈ 0 for a secure cipher algorithm. [17]

The all-zero linear approximation is considered trivial because it holds true regardless of the actual values of the input and output bits. For this reason, assume that we do not consider the all-zero masks in any of the analyses we did in this paper.

In this paper, we mostly use the linear potential (LP) metric instead of correlation. It is calculated as:

$$\text{LP}(\Gamma_{in}, \Gamma_{out}) = (\text{Corr}(\Gamma_{in}, \Gamma_{out}))^2$$

We want the LP value to be close to 0 for a safe cipher algorithm.

Linear Trails and Propagation

Similar to differential cryptanalysis, each round function application affects the input mask. This means we can construct a trail of linear masks resulting from r rounds. Similarly, we can analyze the propagation of masks and list the possible values they can take at certain points in the permutation process. However, unlike differential cryptanalysis, we follow the propagation of an output mask. I.e., we choose an output mask and see how it propagates through the rounds and see what the possible values we can have for the input mask that makes LP ≈ 1. Similar to calculating the approximate value of DP of a differential trail, the LP of a linear trail is calculated as the product of LP values of consecutive linear approximations within a trail. However, unlike DP of a trail, the LP of a trail is not an approximation and is the exact value.

Propagation of Linear Masks through Linear Operations

Similarly to differences, we can derive rules for the propagation of linear masks through linear operations, and similarly to DP values, given an output mask Γ_{out} and an input mask Γ_{in} that results from the effect of a linear transformation λ, it follows that $\text{Corr}(\Gamma_{in}, \Gamma_{out}) = 1$

XOR Property:

For an XOR operation, given an outgoing output mask Γ_{out}, the incoming masks $\Gamma_0, \Gamma_1, \ldots, \Gamma_{n-1}, \Gamma_n$ are all equal to Γ_{out}.

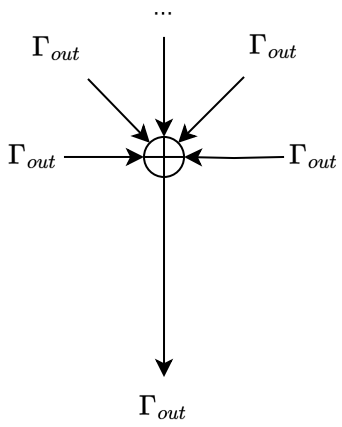


Figure 2.7: Visual representation of the effect of XOR operation on linear masks

Rotation Property:

For rotation operations, the resulting input mask is the output mask rotated in the opposite direction by the same amount. I.e., given an output mask Γ_{out}, the rotation operation n bit to the left would result in the input mask $\Gamma_{out} \ggg n$.

Branch Property:

For a branching operation, given the outgoing masks $\Gamma_0, \Gamma_1, \ldots, \Gamma_{n-1}, \Gamma_n$, the incoming mask is equal to $\bigoplus_{i=0}^{n} \Gamma_i$.

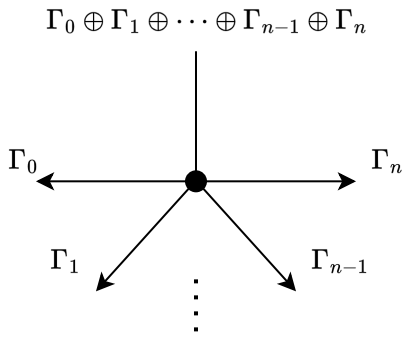


Figure 2.8: Visual representation of the effect of branch operation on linear masks

Generalized Property:

Suppose we have a linear transformation f that modifies an input bitstring and outputs another bitstring. Given the matrix representation M of this linear transformation and the output mask Γ_{out}, the relation between the output mask and input mask Γ_{in} for f is as follows:

$$\Gamma_{in} = M^{\mathrm{T}} \cdot \Gamma_{out}$$

This follows from the fact that

$$\mathrm{Corr}(\Gamma_{in}, \Gamma_{out}) = 1 \text{ if and only if } \Gamma_{in} = M^{\mathrm{T}} \cdot \Gamma_{out} \text{ and } 0 \text{ otherwise}$$

[5]

Note: We do not consider the effect of adding a constant operation, since it is outside the scope of our research.

2.3.3 Algebraic Degree

Algebraic degree analysis is a method used to evaluate and potentially attack cryptographic algorithms by focusing on the algebraic properties of these algorithms, particularly their representations as multivariate polynomial equations.

Monomials

A monomial is an algebraic expression that is a product of variables and a coefficient. For example, $x \cdot y^2$ is a monomial comprised of the variables x and y, and the coefficient 1. This paper focuses on variables whose domain is over $\mathbb{F}_2$. The degree of a monomial whose variables' domain is over $\mathbb{F}_2$ is the number of variables in the expression. For example, $x \cdot y \cdot z$ has a degree of 3.

Over $\mathbb{F}_2$, it is the case that for all Boolean variables x, $x^2 = x$ holds.

Polynomials

Polynomials are a sum of monomials. The degree of a polynomial is the degree of the monomial that has the highest degree. A polynomial is multivariate if there are multiple variables. The variables have a domain in $\mathbb{F}_2$.

Example:

$$P(x, y, z, t, k, l) = xy + z + tkl$$
$$\deg(P(x, y, z, t, k, l)) = \deg(tkl) = 3$$

Vector Boolean Functions

A boolean function is a polynomial from $\mathbb{F}_2^n$ to $\mathbb{F}_2$. A vector boolean function is a function from $\mathbb{F}_2^n$ to $\mathbb{F}_2^m$ where $m > 1$. [3] It can be considered an array of boolean functions.

Example:

$$f(x, y, z) = [x + z + y,\ xyz,\ x + yz]$$

The (algebraic) degree of a vector boolean function is the degree of the boolean function with the highest degree. In the example we gave, it is 3 from the degree of the function xyz.

2.4 Table of Definitions

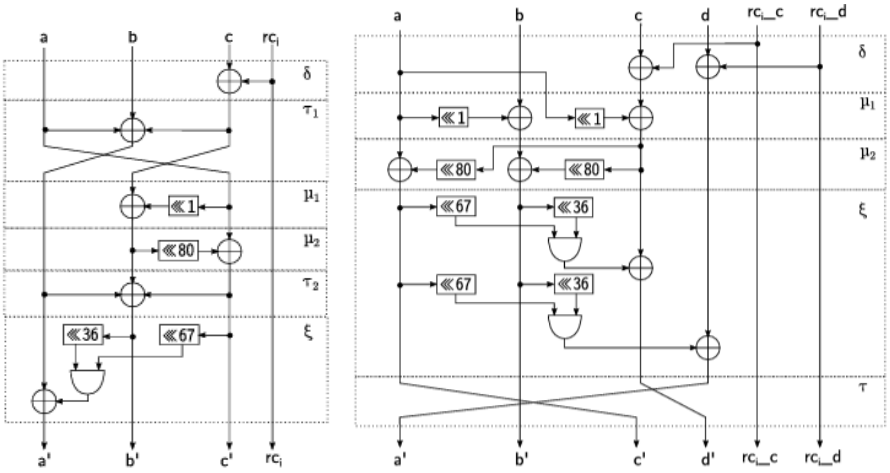
Syntax	Semantics
$\text{len}(x)$	The Number of bits in bitstring x
$x \| y$	Bitstring x is concatenated with the bit string y from the right side.
x^n	a bitstring value made out of repeated concatenation of bitstring x. This pattern is repeated n times.
$\mathcal{P}(S)$	The powerset of S is a set which contains all subsets of set S.
$\log_2(n)$	Logarithm operation in base 2 is applied to a real number n.
$\mathbb{F}_2$	$\mathbb{F}_2 = \{0, 1\}$. More information in Section 2.1.1
$\mathbb{F}_2^n$	The notation $\mathbb{F}_2^n$ denotes the set of all ordered n-tuples over the finite field $\mathbb{F}_2$.
$wt(x)$	Number of bits in bitstring x that have a value of 1.
$\overline{x}$	For each bit in bitstring x, if the bit is 1, then it becomes 0; otherwise it becomes 1.
$x \wedge y$	For each bit position, if at least one of the values in bitstring x and bitstring y is 0, then the resulting bitstring will have 0 in that position. Otherwise, it will have 1.
$x \vee y$	For each bit position, if at least one of the values in bitstring x and bitstring y is 1, then the resulting bitstring will have 1 in that position. Otherwise, it will have 0.
$x \oplus y$	The numbers are added modulo 2 for each bit position in bitstrings x and y.
$x \lll n$	Each bit in bitstring x is shifted left by n bits. If a bit exceeds the left side, then the bit wraps around from the right side of the bitstring. More detailed information is in Section 2.1.4.
$x \ggg n$	Each bit in bitstring x is shifted right by n bits. If a bit exceeds the right side, then the bit wraps around from the left side of the bitstring. More detailed information is in Section 2.1.4.
$\Pr(X = x \mid Y = y)$	The probability of an event x occurring given that event y was observed.

I	A matrix where the diagonals are equal to 1 and the rest of the cells are equal to 0.
M^{T}	The transpose of a matrix M is obtained by swapping its rows and columns. This means that the element at the position (row, column) in M is moved to the position (column, row) in M^{T}. If M is an $n \times m$ matrix, then M^{T} will be an $m \times n$ matrix.
$\mathrm{ord}(M)$	$\mathrm{ord}(M) = n$ where n is the smallest natural number greater than 0 and $M^n = I$. More information is in Section 2.1.3.

Chapter 3

Friet Definition and Concepts

Friet is a duplex-based authenticated encryption scheme built with fault-detection mechanisms. Friet encryption scheme is based on a primitive called Friet-P. It takes a 512-bit long input and yields a 512-bit long output. Its embedding, **Friet-PC**, takes a 384-bit input and yields a 384-bit output.



(a) (b)

Figure 3.1: (a) Round of Friet-PC (b) Round of Friet-P

Friet operates on *states* which are divided into 128-bit long chunks called **limbs**. Friet-PC is comprised of 3 limbs and Friet-P is comprised of 4 limbs. The extra limb and design complexity of Friet-P is because of what is called a parity limb, which is used to satisfy so-called parity equations. The first three limbs are called native limbs.

The values rc_i, rc_i_c, and rc_i_d are not additional limbs. They are round constants. For each round, an XOR operation is carried out with the corresponding round constant.

A *limb adaptation* is an operation that modifies a native limb, by bitwise adding to it a function ϕ of the state. It also adds the function ϕ to each parity limb that depends

on that same native limb.

A *limb transposition* operation is a reordering of limbs, with a possible correcting adaptation to leave the parity equations invariant. A native limb transposition refers to two native limbs being swapped. A non-native limb transposition refers to a native limb swapping places with a parity limb. Furthermore, the operations are divided into different step functions, which are small operations that make up Friet-PC. They are denoted by Greek letters. The explanation of each step function are as follows:

- τ_1 and τ_2 are non-native limb transpositions,
- δ is a round constant addition that is a limb adaptation,
- two mixing steps μ_1 and μ_2 that are limb adaptations,
- a non-linear step ξ, also a limb adaptation

Friet-PC:

$$\begin{aligned} c &\leftarrow c \oplus rc_i && \delta_i \\ (a,b,c) &\leftarrow (a \oplus b \oplus c, c, a) && \tau_1 \\ b &\leftarrow b \oplus (c \lll 1) && \mu_1 \\ c &\leftarrow c \oplus (b \lll 80) && \mu_2 \\ (a,b,c) &\leftarrow (a, a \oplus b \oplus c, c) && \tau_2 \\ a &\leftarrow a \oplus ((b \lll 36) \wedge (c \lll 67)) && \xi \end{aligned} \tag{3.1}$$

The focus of this paper is on the Friet-PC round function. We index the limbs of Friet-PC as limbs 1, 2, and 3. In Figure 3.1 (a), these would correspond to segments of a, b and c. [14]

Chapter 4

The Weaknesses of Friet-PC

In [17], it is shown that the weakness of the encryption scheme arises from insufficient security against differential and linear cryptanalysis. Crucially, these attacks are highly effective in the primitive of the construction. This weakness propagates through every round of permutation, making the scheme itself insecure against these statistical attacks.

It is showcased in [17], that for certain r-round differentials and r-round linear approximations, the DP and the LP are both equal to 1. The conditions for this to hold for differences and linear masks are defined in Conditions (4.1) and (4.2).

Differential Condition 1 [17]: The DP of a non-zero differential trail $(\alpha, \beta, \gamma) \rightarrow (\alpha', \beta', \gamma') \rightarrow (\alpha'', \beta'', \gamma'')$ for the 2-round Friet-PC is 1 if and only if

$$\begin{aligned} \alpha = \alpha' = \alpha'' &= \mathbf{1}^{128} \\ \beta = \beta' = \beta'' &= \mathbf{0}^{128} \\ \gamma = \gamma' = \gamma'' &= \mathbf{0}^{128} \end{aligned} \tag{4.1}$$

Linear Condition 1 [17]: For the input linear mask $\Gamma_{in} = (\mathbf{0}^{128}, \mathbf{0}^{128}, \mathbf{1}^{128})$ and output linear mask

$$\Gamma_{out} = (\mathbf{0}^{128}, \mathbf{0}^{128}, \mathbf{1}^{128}), \tag{4.2}$$

$\mathrm{LP}(\Gamma_{in}, \Gamma_{out})$ for n-round Friet-PC is 1, where $n \geq 1$.

This means the Friet-PC in Even-Mansour construction is distinguishable from a random permutation with an advantage of ≈ 1. In the following paragraph, we showed this by defining two distinguishers between the Random Permutation $\mathcal{RP}$ and the primitive Friet-PC under a two-key Even–Mansour scheme; one for the differential case, and one for the linear. We constructed a differential distinguisher $\mathcal{D}_1$ as follows:

1. Choose a plaintext $P_0 = p_0 \| p_1 \| p_2$ and generate plaintext $P_1 = \overline{p_0} \| p_1 \| p_2$.

2. Apply the provided permutation on plaintexts P_0 and P_1 and get the associated ciphertext pair $C_0 = c_0 \| c_1 \| c_2$ and $C_1 = c_0^* \| c_1^* \| c_2^*$.

3. If $(c_0^*, c_1^*, c_2^*) = (\overline{c_0}, c_1, c_2)$, return 1; else, return 0.

We also constructed a linear distinguisher $\mathcal{D}_2$ as follows:

1. Choose n plaintexts in the form of $P = p_0 \| p_1 \| p_2$.

2. Apply the provided permutation on plaintexts P and get the associated ciphertexts in the form of $C = c_0 \| c_1 \| c_2$.

3. If $wt(c_2) = wt(p_2) \pmod 2$ for all plaintext-ciphertext pairs, store 1; else, store 0.

4. Check whether all return values are 1, if so, the distinguisher returns 1, else, the distinguisher returns 0.

The distinguisher returns 1 when it suspects that it is applying permutation on plaintexts using Friet-PC. It returns 0 when it suspects that it is applying permutation on plaintexts using the random permutation ($\mathcal{RP}$).

To calculate the advantage the distinguisher has, we used the advantage formula from Section 2.2.5.

We first calculated the advantage of the differential distinguisher, and then the linear distinguisher.

The probability of $\mathcal{D}_1$ returning 1, given it is using Friet-PC is 1. The probability of $\mathcal{D}_1$ returning 1, given it is using $\mathcal{RP}$ is 2^{-384}. This is because the probability of $(c_1^*, c_2^*, c_3^*) = (\overline{c_1}, c_2, c_3)$ is affected by all three 128-bit limbs since each tag must match a specific 128-bit value for the condition to hold. I.e., $\Pr(c_1^* = \overline{c_1}) \cdot \Pr(c_2^* = c_2) \cdot \Pr(c_3^* = c_3) = 2^{-128} \cdot 2^{-128} \cdot 2^{-128} = 2^{-384}$. The advantage is thus $Adv_{\mathcal{D}_1} = 1 - 2^{-384} \approx 1$.

The probability of $\mathcal{D}_2$ returning 1, given it is using Friet-PC is 1. The probability of $\mathcal{D}_2$ returning 1, given it is using $\mathcal{RP}$ and n plaintexts were chosen, is 2^{-n}. Using the above formula, we get $Adv_{\mathcal{D}_2} = 1 - 2^{-n}$. For bigger values of n, the advantage is equal to ≈ 1. After repeated applications of Friet-PC for several rounds, the DP of the differential trail and the LP of the linear trail are 1 because the input difference and output masks propagate through Friet-PC without changing. With this weakness in Friet-PC, it is not feasible to construct an encryption scheme that has at least 128 bits of security. This can be shown by using the security strength formula from Section 2.2.7 where M is the number of online queries and N is the number of offline queries.

Security against differential distinguisher:

$$\log_2 \left(\frac{M+N}{Adv_{\mathcal{D}_1}} \right) \approx \log_2 \left(\frac{2+0}{1} \right) = 1$$

Therefore, we have 1 bit of security against differential distinguishing attacks.

Security against linear distinguisher:

$$\log_2\left(\frac{M+N}{Adv_{\mathcal{D}_2}}\right) \approx \log_2\left(\frac{M+0}{1-2^{-M}}\right) \overset{M=1000}{=} \log_2\left(\frac{1000}{1-2^{-1000}}\right) \approx 9.97$$

Theoretically, Friet-PC constructed with Even-Mansour should have $\log_2(2^{\frac{384}{2}}) = \log_2(2^{192}) = 192$ bits of security according to [4]. Since both distinguishers have security bounds lower than 192, the permutation is not secure against linear and differential distinguishing attacks.

Less critically, but still importantly, in [17], the conditions for when the DP and the LP are equal to 1 for a single round are highlighted.

Differential Condition 2 [17]: The differential probability of trail $(\alpha, \beta, \gamma) \rightarrow (\alpha', \beta', \gamma')$ for the i-th round function is 1 if and only if

$$\begin{aligned} \alpha' &= \alpha \oplus \beta \oplus \gamma \\ \alpha \oplus (\alpha \lll 1) \oplus \beta &= \mathbf{0}^{128} \\ \alpha \oplus (\alpha \lll 81) \oplus (\gamma \lll 80) &= \mathbf{0}^{128} \\ \beta' &= \mathbf{0}^{128} \\ \gamma' &= \mathbf{0}^{128} \end{aligned} \tag{4.3}$$

Simply put, there exists a differential with DP 1 where the second and third limbs of the output difference are the all-zero bitstrings.

Linear Condition 2 [17]: Let $\Gamma_{in} = (\alpha, \beta, \gamma)$ and $\Gamma_{out} = (\alpha', \beta', \gamma')$ be the input and output linear masks of the i-th round function. The absolute value of the correlation $\mathrm{Corr}(\Gamma_{in}, \Gamma_{out})$ is 1 if and only if

$$\begin{aligned} \alpha' &= \mathbf{0}^{128} \\ \alpha \oplus (\beta' \ggg 1) \oplus \gamma' \oplus ((\beta' \oplus \gamma') \ggg 81) &= \mathbf{0}^{128} \\ \beta \oplus \beta' &= \mathbf{0}^{128} \\ \gamma \oplus ((\beta' \oplus \gamma') \ggg 80) &= \mathbf{0}^{128} \end{aligned} \tag{4.4}$$

In other words, there exists a pair of input-output masks such that their LP is equal to 1, where the limb 1 output mask is the all-zero bitstring.

Chapter 5

Strategy of Fixing Friet-PC

We tried to come up with alternative designs by first focusing on the differential propagation weakness. Afterward, we checked whether the same proposal also resolved the linear propagation weakness. If it did not, then we repeated the process. We came up with a few initial proposals that we thought would mitigate this issue. We went into more detail in Section 6. These proposals were engineered with computational resourcefulness and preserving the fault detection characteristics of the original primitive in mind.

In our initial attempts to fix Friet-PC (3.1), we tried to keep its initial design rationale. In other words, we wanted to keep the effects and order of limb transpositions and limb adaptation step functions. So for example, the new equation we define in a limb transposition step function should also be another limb transposition step function.

In addition to keeping the order of the limb operations, we wanted to keep the bitwise AND operation as our only non-linear operation. Finally, we aimed to limit the usage of additional XOR and rotation operations as much as possible, ideally reducing the number of bitwise operations or keeping the same as the original design. This way, the software implementation of our fix could be at least as computationally effective as the original design.

Chapter 6

Unsuccessful Attempts at Fixing the Primitive

At the start of the research, we came up with three candidates. Ultimately, these candidates turned out to be insecure when we tested them for differential and linear cryptanalysis with the methodology outlined in Section 6.4. We discuss the reasons why they failed in more detail in Section 6.5. In the following sections, we talked about the specifications of each attempt and what our reasoning was when coming up with these specifications.

6.1 Attempt 1

Initially, we came up with the candidate shown in Figure 6.1. We first focused on the reasons for the structural issues that were causing the problem in Condition (4.1). We concluded that the reason why the limb 1 difference did not change when it was $\mathbf{1}^{128}$ was due to a lack of limb transposition/adaptation on limb 1 that could alter the difference. The fact that the first limb difference α retained the value $\mathbf{1}^{128}$ contributed to the reason why limb 2 and 3 differences did not change when $\mathbf{0}^{128}$. This was because $\alpha = \alpha \lll 81 = \mathbf{0}^{128}$ and $\alpha = \alpha \lll 1 = \mathbf{0}^{128}$. So in μ_2, limb 3 became $\mathbf{0}^{128}$ and in τ_2, limb 2 became $\mathbf{0}^{128}$. Hence, all three limb differences stayed the same.

We then figured that switching the differences of limb 1 and 2 during the second non-native limb transposition τ_2 would ensure that the first limb is modified, while also modifying the difference of limb 2 to a non-zero difference.

The proposed round function is as follows:

$$\begin{aligned}
c &\leftarrow c \oplus rc_i \qquad \delta_i \\
(a, b, c) &\leftarrow (a \oplus b \oplus c, c, a) \qquad \tau_1 \\
b &\leftarrow b \oplus (c \lll 1) \qquad \mu_1 \\
c &\leftarrow c \oplus (b \lll 80) \qquad \mu_2 \\
\boldsymbol{(a, b, c)} &\leftarrow \boldsymbol{(a \oplus b \oplus c, a, c)} \qquad \boldsymbol{\tau_2} \\
a &\leftarrow a \oplus ((b \lll 36) \wedge (c \lll 67)) \qquad \xi
\end{aligned} \tag{6.1}$$

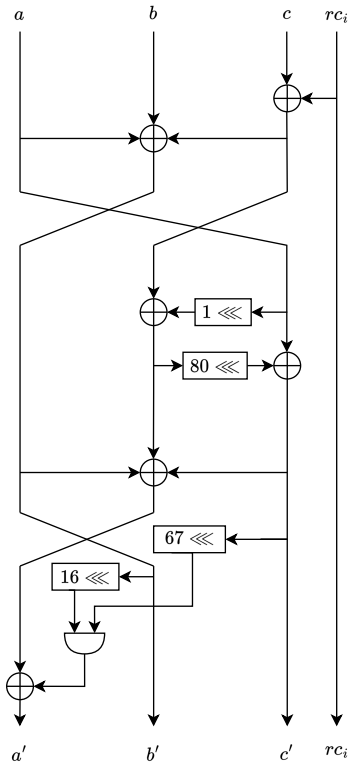


Figure 6.1: First attempt

This design not only conserves the limb operation structure and order of the original Friet-PC, but it is also the result of minimal change in the original design. Another advantage is that the change can be implemented easily in the hardware by switching wire connections. This attempt failed due to weaknesses against linear cryptanalysis.

6.2 Attempt 2

Similar to the first candidate, we aimed to have some kind of operation that modifies limb 1. In this case, we figured that there was an abundance of operations applied on the third limb. So we moved the first rotate-XOR operation in μ_1 to limb 1, as depicted in Figure 6.2. Doing so ensured that the difference of limb 2 became non-zero while modifying the difference of limb 1.

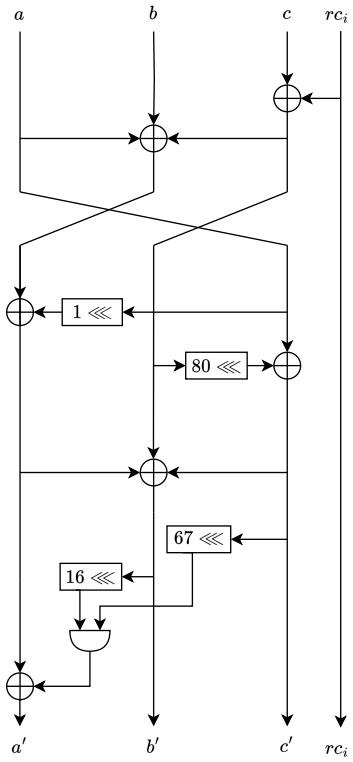


Figure 6.2: Second attempt

The proposed round function is as follows:

$$\begin{aligned} c &\leftarrow c \oplus rc_i \qquad \delta_i \\ (a,b,c) &\leftarrow (a \oplus b \oplus c, c, a) \qquad \tau_1 \\ \boldsymbol{a} &\leftarrow \boldsymbol{a \oplus (b \lll 1)} \qquad \boldsymbol{\mu_1} \\ c &\leftarrow c \oplus (b \lll 80) \qquad \mu_2 \\ (a,b,c) &\leftarrow (a, a \oplus b \oplus c, c) \qquad \tau_2 \\ a &\leftarrow a \oplus ((b \lll 36) \wedge (c \lll 67)) \qquad \xi \end{aligned} \tag{6.2}$$

This proposal preserves the amount of resources like the first attempt, but unlike the first attempt, this modification is made to a limb adaptation process. Unfortunately, this proposal is also weak against linear cryptanalysis.

6.3 Attempt 3

This proposal is a natural expansion of the first two attempts that were insecure against linear cryptanalysis. We assumed the combined effects of the changes from the previous attempts, as shown in Figure 6.3, would circumvent the linear cryptanalysis weakness that the previous two attempts faced.

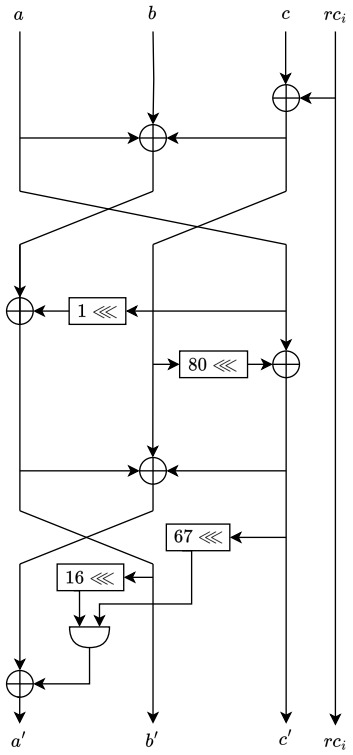


Figure 6.3: Third attempt

The proposed round function is as follows:

$$\begin{aligned}
c &\leftarrow c \oplus rc_i \qquad \delta_i \\
(a,b,c) &\leftarrow (a \oplus b \oplus c, c, a) \qquad \tau_1 \\
\boldsymbol{a} &\leftarrow \boldsymbol{a \oplus (b \lll 1)} \qquad \boldsymbol{\mu_1} \\
c &\leftarrow c \oplus (b \lll 80) \qquad \mu_2 \\
\boldsymbol{(a,b,c)} &\leftarrow \boldsymbol{(a \oplus b \oplus c, a, c)} \qquad \boldsymbol{\tau_2} \\
a &\leftarrow a \oplus ((b \lll 36) \wedge (c \lll 67)) \qquad \xi
\end{aligned} \tag{6.3}$$

6.4 Testing the Quality of the Candidates

We mentioned the relations that cause the high DP and LP values the current design has in Conditions (4.1), (4.2), (4.3) and (4.4). So for each attempt to fix the permutation, we tested various input differences and output masks. Our goal was to identify differences or masks that could form trails with a DP or LP of 1 over multiple rounds.

We wrote two simple algorithms Difference-Evaluation and Mask-Evaluation shown in Algorithm 1 and Algorithm 3 to check the security of each proposal. The first algorithm we wrote was Algorithm 1. It uses Algorithm 2 to look for input differences $\Delta_{in} = \alpha \| \mathbf{0}^{128} \| \mathbf{0}^{128}$ such that $\mathrm{DP}(\Delta_{in}, \Delta_{out}) = 1$ for an i-round differential where Δ_{out}

is the output difference and $i > 1$. As we know from Condition (4.3), DP is 1 for a single round differential if limb 2 and limb 3 of the output difference is $\mathbf{0}^{128}$, I.e., the input difference to the bitwise AND operation is $\mathbf{0}^{256}$. So the algorithm looks for an input difference where the effects of the linear operations on it result in a difference with limb 2 and limb 3 equal to $\mathbf{0}^{128}$ respectively. We denote the linear section of the round function as $F_L = \tau_2 \circ \mu_2 \circ \mu_1 \circ \tau_1$. We ignore δ_i since it does not affect the values of the input differences/masks. Algorithm 1 tries all the bitstrings α of length 2^n where $1 \leq n \leq 5$ for the limb 1 difference. We can analyze equations with bitstrings that have a length of less than 128 bits because we can interpret short bitstrings as a pattern that makes out a complete 128-bit long bitstring for the limb. This and other algorithms we mention later on take advantage of the properties of repeating bit patterns mentioned in Section 2.1.4. For example, examining the input $\mathbf{10}\|\mathbf{0}\|\mathbf{0}$ is the same thing as examining the bitstring $(\mathbf{10})^{64}\|\mathbf{0}^{128}\|\mathbf{0}^{128}$, since the first variant is the reduced version of the latter.

Algorithm 1 DIFFERENCE-EVALUATION

```
1: function DIFFERENCE_EVALUATION(F_L)
2:     for size_alpha from 2 to 32, doubling each iteration do
3:         for each bitstring α of size size_alpha do
4:             DT(α, F_L)
5:         end for
6:     end for
7: end function
```

Algorithm 2 (DT) tests the input difference $(\alpha, \mathbf{0}^n, \mathbf{0}^n)$ with F_L where $n = \text{len}(\alpha)$. If Algorithm 2 evaluates the limb 2 and limb 3 output differences as equal to $\mathbf{0}^n$, then DP is 1 for at least one round differential. Thus, this input can be dangerous for > 1 round differential. If the input and output limb 1 differences are the same, then this implies the same problem as Condition (4.1), DP is 1 for an i-round differential where $i \geq 1$. Thus, this input is very dangerous.

An important thing to note is, even though we mentioned the output difference is the one that should have its last two limbs the all-zero bitstring, the algorithm DT tries different values $(\alpha, \mathbf{0}^{128}, \mathbf{0}^{128})$ for the input difference instead. The reason for doing this is because we want to catch differentials with DP equal to 1 that may propagate n rounds like the one we saw in Condition (4.1). A case like this could happen if and only if the last two limbs of both the input and the output difference are the all-zero bitstrings. Moreover, since we follow the propagation of differential from the input difference to the output difference, it is more convenient for us to try values for limb 1 input difference.

Algorithm 3 tests against linear cryptanalysis. It uses Algorithm 4 to check for output masks $\Gamma_{out} = \mathbf{0}^{128}\|\beta'\|\gamma'$ such that $\text{LP}(\Gamma_{in}, \Gamma_{out}) = 1$ for i-round linear approximation where Γ_{in} is the input mask and $i > 1$. LP is equal to 1 for masks propagating through linear operations. The only non-linear operation is bitwise AND. According to the correlation conditions for bitwise AND operation mentioned in [17], if the output mask

Algorithm 2 DT

```
1: function DT(init_alpha, F_L)
2:     n ← len(α)
3:     (α, β, γ) ← (init_alpha, 0^n, 0^n)
4:     (α', β', γ') ← F_L(α, β, γ)
5:     if α' ≠ 0^n AND β' = 0^n AND γ' = 0^n then
6:         if α' = α then
7:             print(α ,"is very dangerous!")
8:         else
9:             print(α ,"can be dangerous.")
10:        end if
11:    end if
12: end function
```

is all-zero, then the LP with the input mask for the bitwise AND operation is equal to 1 iff the input mask is also all-zero. The algorithm finds output masks $\Gamma_{out} = \mathbf{0}^{128} \| \beta' \| \gamma'$, that result in the input mask $\Gamma_{in} = \mathbf{0}^{128} \| \beta \| \gamma$. Algorithm 3 tries all the bitstrings of length 2^m where $1 \leq m \leq 3$ for the masks of limbs 2 and 3.

Algorithm 3 MASK-EVALUATION

```
1: function MASK_EVALUATION(dangerous_input, dangerous_output, F_L)
2:     for size_beta from 1 to 8, doubling each iteration do
3:         for size_gamma from 1 to 8, doubling each iteration do
4:             for each bitstring β of size size_beta do
5:                 for each bitstring γ of size size_gamma do
6:                     (dangerous_input, dangerous_output)
                           ← LT(β, γ, dangerous_input, dangerous_output, F_L)
7:                 end for
8:             end for
9:         end for
10:    end for
11: end function
```

The reason why we tried bitstrings only for limb 1 in the differential case and we tried bitstrings for limbs 2 and 3 in the linear case is that for a dangerous output difference, the differences between limbs 2 and 3 are all-zero, this means the only varying part is the limb 1. Thus, we only tried differences for a single limb. For a dangerous output mask, limb 1 is all-zero, this means that the other two limbs are the varying parts of the bitstring. Thus, we tried masks for two limbs instead of one. The upper bound for what the longest bitstring pattern should be for both algorithms was chosen to test small-bit patterns instead of big ones.

Algorithm 4 (LT) tests the output mask $(\mathbf{0}^n, \beta, \gamma)$ with F_L where $n = \max(\text{len}(\gamma), \text{len}(\beta))$. If LT evaluates the limb 1 input mask as $\mathbf{0}^n$, then LP is 1 for at least one round linear approximation, therefore the output mask is added to the set of dangerous outputs and the input mask is added to the set of dangerous inputs of the algorithm. If the input and output limb 2 and limb 3 masks are the same, then this implies the same problem as Condition (4.2), LP is 1 for an i-round linear approximation where $i \geq 1$. Thus, this output mask is very dangerous.

However, not all i-round linear trails have to be made up of the same mask for the LP of the trail to be 1. The LP of a trail would be 1 if each of the masks in a trail has an all-zero mask in limb 1. We wanted to consider the simplest case in which this would happen. This is why, given the dangerous output mask Γ_{out} and the input mask Γ_{in} which was calculated as the effect of application of F_L, the algorithm checks whether Γ_{out} has already been an input mask for Γ_{in}.

That is to say, given the masks $\Gamma_1 = \mathbf{0}^{128} \| \beta \| \gamma$ and $\Gamma_2 = \mathbf{0}^{128} \| \beta' \| \gamma'$, if $\text{LP}(\Gamma_1, \Gamma_2) = 1$ and $\text{LP}(\Gamma_2, \Gamma_1) = 1$ for a single round linear approximation, then $\text{LP}(\Gamma_1, \Gamma_2) = 1$ and $\text{LP}(\Gamma_2, \Gamma_1) = 1$ for an odd number of rounds linear approximation.

Algorithm 4 LT

function LT($init\_beta, init\_gamma, dangerous\_input, dangerous\_output, F_L$)
  $n \leftarrow \max(\text{len}(\beta'), \text{len}(\gamma'))$
  $(\alpha', \beta', \gamma') \leftarrow (\mathbf{0}^n, init\_beta, init\_gamma)$
  if $\beta' = 0$ AND $\gamma' = 0$ **then**
    return $(dangerous\_input, dangerous\_output)$
  end if
  Declare (α, β, γ)
  if $\text{len}(\beta') \geq \text{len}(\gamma')$ **then**
    $length\_factor \leftarrow \frac{n}{\text{len}(\gamma')}$
    $(\alpha', \beta', \gamma') \leftarrow (\alpha', \beta', (\gamma')^{length\_factor})$
    $(\alpha, \beta, \gamma) \leftarrow F_L(\alpha', \beta', \gamma')$
  else
    $length\_factor \leftarrow \frac{n}{\text{len}(\beta')}$
    $(\alpha', \beta', \gamma') \leftarrow (\alpha', (\beta')^{length\_factor}, \gamma')$
    $(\alpha, \beta, \gamma) \leftarrow F_L(\alpha', \beta', \gamma')$
  end if
  if $\alpha = \mathbf{0}^n$ AND ($\beta \neq \mathbf{0}^n$ OR $\gamma \neq \mathbf{0}^n$) **then**
    $dangerous\_output.insert((\alpha', \beta', \gamma'))$
    $dangerous\_input.insert((\alpha, \beta, \gamma))$
    $containment\_condition \leftarrow dangerous\_output.contains((\alpha, \beta, \gamma))$ AND $dangerous\_input.contains((\alpha', \beta', \gamma'))$
    if $\gamma = \gamma'$ AND $\beta = \beta'$ **then**
      print((β', γ'),"pair is very dangerous!")
    else if $containment\_condition$ **then**
      print("The pairs ", (β', γ'), "and ", (β, γ)"are referencing each other!")
    else
      print((β', γ'),"can be dangerous")
    end if
  end if
  return $(dangerous\_input, dangerous\_output)$
end function

The variables in the pseudocodes are explained as follows:

- F_L : The linear part of the round function we are testing against differential and linear cryptanalysis.
- *dangerous_output*: The set of output masks $\Gamma' = (\mathbf{0}^{128}, \beta', \gamma') \in \mathbb{F}_2^{3n}$, where the effect of the round function on Γ' results in an input mask $\Gamma = (\mathbf{0}^{128}, \beta, \gamma)$.
- *dangerous_input*: The set of input masks $\Gamma = (\mathbf{0}^{128}, \beta, \gamma) \in \mathbb{F}_2^{384}$, where the effect of the round function on some output mask $\Gamma' = (\mathbf{0}^{n}, \beta', \gamma')$ results in Γ.

  dangerous_output.insert (or *dangerous_input.insert*) inserts a mask with each limb's bit pattern representing the mask for that limb expanded to 128 bits. For example,

  $dangerous\_output.insert((\mathbf{10}, \mathbf{0}, \mathbf{1})) = dangerous\_output.insert(((\mathbf{10})^{64}, \mathbf{0}^{128}, \mathbf{1}^{128}))$

  dangerous_output.contains(Γ) (or *dangerous_input.contains*(Γ)) checks whether a mask Γ (each limb mask expanded to 128 bits) is in the set.

Normally, for Algorithm 2 a good conscious check would be to do a similar thing we do in Algorithm 4, i.e., keep track of differences that have been found dangerous in the input and the output, then check whether the input difference we are trying is found in the set of dangerous outputs and vice versa. This would make it possible for us to construct a dependency graph and see whether we will always have the (non-zero, $\mathbf{0}^{128}, \mathbf{0}^{128}$) structure. It turned out we did not need to consider this for the few proposals we came up with since none of the fixes had a pair of input-output differences with a similar structure.

Notice that in the pseudocode for Algorithm 4 on lines 11 and 15, the round function is applied after the lengths of repeating patterns are equalized. This ensures that the XOR operations are done on bit strings with the same pattern frequency.

Example:

$$(\mathbf{01})^{64} \oplus \mathbf{1}^{128} = (\mathbf{01})^{64} \oplus (\mathbf{1}^2)^{64} = (\mathbf{01})^{64} \oplus (\mathbf{11})^{64} = (\mathbf{10})^{64}$$

In the end, these algorithms revealed the shortcomings of each attempt we made.

6.5 The Design Flaw of the Initial Attempts

By the end of our analysis, we found out that all attempts displayed improvement against differential cryptanalysis. The reason for this is that the alterations not only modify limb 1 for an input in the form $(\alpha, \mathbf{0}^{128}, \mathbf{0}^{128})$, they also modify the values of the other limbs to a non-zero value, thus preventing the risk of DP value 1.

However, it turned out that neither one of the three candidates is safe against linear cryptanalysis. Below, we highlight the inputs that this weakness arises.

Given the output mask $(\mathbf{0}^{128}, \beta', \gamma')$, the input mask is represented for each candidate as follows:

The input mask equation for the first candidate.

$$\begin{aligned} \alpha &= \beta' \oplus \gamma' \oplus (\gamma' \ggg 81) \\ \beta &= \beta' \\ \gamma &= \beta' \oplus (\gamma' \ggg 80) \end{aligned} \tag{6.4}$$

The input mask equation for the second candidate.

$$\begin{aligned} \alpha &= \gamma' \oplus (\beta' \ggg 1) \\ \beta &= \beta' \\ \gamma &= \beta' \oplus (\beta' \ggg 80) \oplus (\gamma' \ggg 80) \end{aligned} \tag{6.5}$$

The input mask equation for the third candidate.

$$\begin{aligned} \alpha &= \beta' \oplus \gamma' \oplus (\beta' \ggg 1) \\ \beta &= \beta' \\ \gamma &= \beta' \oplus (\gamma' \ggg 80) \end{aligned} \tag{6.6}$$

For the first candidate, the output mask $\Gamma_{out} = \mathbf{0}^{128} \| \mathbf{0}^{128} \| \mathbf{1}^{128}$, results in the input mask $\Gamma_{in} = \mathbf{0}^{128} \| \mathbf{0}^{128} \| \mathbf{1}^{128}$. Since the input-output mask pair is equal to each other, this output mask persists throughout the entire series of round function applications. Thus, $\mathrm{LP}(\Gamma_{in}, \Gamma_{out}) = 1$.

For the second candidate, the output mask $\Gamma_{out} = \mathbf{0}^{128} \| \mathbf{1}^{128} \| \mathbf{1}^{128}$, results in the input mask $\Gamma_{in} = \mathbf{0}^{128} \| \mathbf{1}^{128} \| \mathbf{1}^{128}$. Similarly to the first candidate, this implies $\mathrm{LP}(\Gamma_{in}, \Gamma_{out}) = 1$.

The third candidate did not have an output mask with a repeating bit pattern of length less than 16 that persists. However, for the output mask $\mathbf{0}^{128} \| (\mathbf{01})^{64} \| \mathbf{1}^{128}$ we end up with the input mask $\mathbf{0}^{128} \| (\mathbf{01})^{64} \| (\mathbf{10})^{64}$. If we then have this mask as the output mask for the next round, we end up with the input mask $\mathbf{0}^{128} \| (\mathbf{01})^{64} \| \mathbf{1}^{128}$. After two rounds, we ended up with the same mask as we started. Since the limb 1 mask in both masks is all zero, the Toffoli Stream operation does not exhibit a non-linear effect on the masks. I.e., the propagation of the output mask is non-probabilistic. Thus, the LP for the input-output mask pair $(\mathbf{0}^{128} \| (\mathbf{01})^{64} \| \mathbf{1}^{128}, \mathbf{0}^{128} \| (\mathbf{01})^{64} \| \mathbf{1}^{128})$ for an even number of rounds is 1. Equivalently, the LP for the input-output mask pair $(\mathbf{0}^{128} \| (\mathbf{01})^{64} \| (\mathbf{10})^{64}, \mathbf{0}^{128} \| (\mathbf{01})^{64} \| \mathbf{1}^{128})$ for an odd number of rounds is 1.

The main and common flaw in these candidates is that the limb 2 mask propagates as it is. This means that if we fix a value for β', it becomes easy to find a value for γ' such that $\alpha = \mathbf{0}^{128}$. This paves the way for effective linear cryptanalysis.

In the next section, we begin our analysis of whether a suitable adjustment exists to the linear operations such that it is reasonably secure against differential and linear cryptanalysis.

Chapter 7

Investigation of Suitable Linear Transformations

We have proposed three potential fixes and we have found out that all three of them failed critically due to the existence of iterative linear approximation with LP equal to 1. In this section, we investigated the existence of a proper fix to the primitive with the following conditions:

1. Combinations $\{(x, \mathbf{0}^{128}, \mathbf{0}^{128}) \in \mathbb{F}_2^{384} \setminus \{(\mathbf{0}^{128}, \mathbf{0}^{128}, \mathbf{0}^{128})\}\}$ do not appear frequently in a differential trail.
2. Combinations $\{(\mathbf{0}^{128}, x, y) \in \mathbb{F}_2^{384} \setminus \{(\mathbf{0}^{128}, \mathbf{0}^{128}, \mathbf{0}^{128})\}\}$ do not appear frequently in a linear trail.
3. The only non-linear operation is a single application of a Toffoli Stream operation.

We defined what we mean by "frequently" formally in the next section. A single round can be divided into two sections. The first section is the linear section which is a series of linear operations on three limbs and the round constant addition. The second is a non-linear section, which is a bitwise AND operation on limbs 2 and 3, and an XOR operation with the result of the bitwise AND operation on limb 1.
Any linear operation on a vector space V can be represented as a square matrix of size equal to the dimension of V.

This interpretation of the permutation like in Figure 7.1, together with the requirement of a single AND operation, reduced the problem of fixing Friet-PC into a problem of finding a good matrix that represents the linear operation.

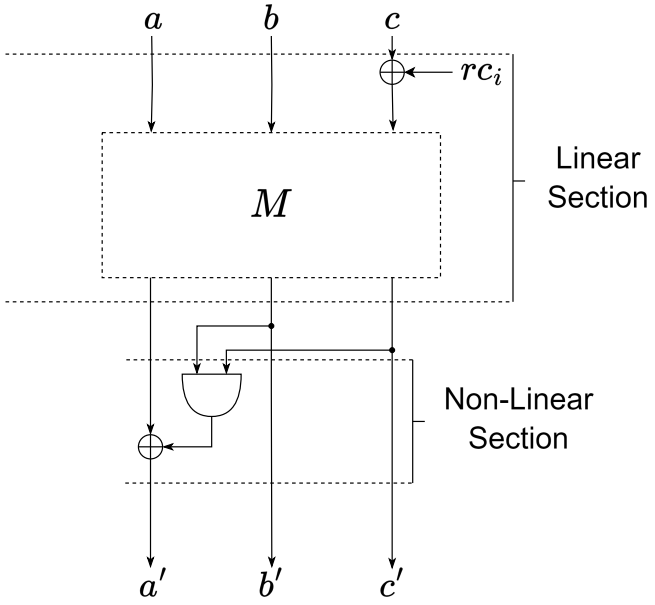


Figure 7.1: Sections of a permutation

7.1 What Makes a Matrix Suitable

Although we could represent linear operations as matrices, we wanted to consider a sub-space of matrices, as not all matrices fit the criteria we look for. Given that each limb has 128 bits, we would ideally look for a 384×384 matrix that fits these criteria. However, in a naive search, there are $2^{384 \cdot 384}$ different unique matrices we would need to consider. Since we want to construct a permutation, the matrix has to be invertible. A matrix is invertible if its columns are linearly independent. The matrices that fit this criterion are the matrices that form the group $GL(n, 2)$ where n is the dimension of the square matrix.

Although it is trivial to find a linear transformation that is invertible as the only criterion, we enforced additional restrictions on the matrices to make them suitable as a linear round. The first restriction we considered is to ensure that: $(\alpha', \mathbf{0}^{128}, \mathbf{0}^{128}) \neq M \cdot (\alpha, \mathbf{0}^{128}, \mathbf{0}^{128})$ for an input difference $(\alpha, \mathbf{0}^{128}, \mathbf{0}^{128})$ and an output difference $(\alpha', \mathbf{0}^{128}, \mathbf{0}^{128})$ where $\alpha \neq \mathbf{0}^{128}$ and $\alpha' \neq \mathbf{0}^{128}$. In the absence of this restriction, the AND operation returns the same output difference. The second restriction was to ensure the LP is not 1. Suppose applying the transpose matrix M^T $\text{ord}(M)$ times on a non-all-zero output mask $(\mathbf{0}^{128}, \beta, \gamma)$ results in linear trail T. We want to ensure that:

$$\exists(\alpha, \beta, \gamma) \in T, [\alpha \neq \mathbf{0}^{128}]$$

This restriction is to ensure that in the permutation constructed from this matrix, LP is not equal to 1 for each iterative linear approximation in the linear trail. As we mentioned

in Section 6.4, the LP for a linear approximation for the bitwise AND operation is equal to 1 when the output mask is all-zero. This corresponds to the mask of the first limb. If there is then a trail where each of the limb 1 masks is all-zero, then the LP of the entire trail is 1, so we wished to avoid this problem and chose the matrices with this in mind.

With these restrictions, it was still unfeasible to compute and check 384×384 matrices. However, we reduced our search space with the intuition that, any matrix which did not work for shorter repeating patterns of bits, would not work for longer repeating patterns. Similar to binary operations on bitstrings made of repeating bit patterns, a matrix applied on a vector with repeating patterns is equivalent to applying a matrix with reduced dimensions to that pattern. We defined the reduction operation of a matrix to a lower dimension as follows:

Suppose we have a circulant matrix of size $2n \times 2n$
We take the first column/row and divide it into two $1 \times n$ vectors.
We add the two vectors together and from the resulting vector $\vec{v}$, we create an $n \times n$ circulant matrix.

Example: Suppose we have the following matrix:

$$M = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

Take the following vector:

$$\vec{v} = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

We have $\vec{w}$:

$$\vec{w} = M \cdot \vec{v} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

The circulant matrix M is formed from the vector $(1, 0, 0, 1)$. If we follow the reduction process we outlined, the reduced matrix will be made up of vector $(1, 0) \oplus (0, 1) = (1, 1)$. Thus, our new matrix is:

$$M' = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

If we then reduce $\vec{v}$ into its repeating pattern, we get:

$$\vec{v'} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Now

$$\vec{w}' = M' \cdot \vec{v}' = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

As can be seen, this is indeed the reduced version of the vector $\vec{w} = (1, 1, 1, 1)$.

If we started our search with a single-bit pattern repeating 128 times per limb, we would only need to consider valid 3×3 matrices. These matrices form $GL(3, 2)$. We determined the number of invertible matrices as the formula mentioned in Section 2.1.3 for general linear groups over finite fields. So there are

$$\prod_{k=0}^{2} (2^3 - 2^k) = 7 \cdot 6 \cdot 4 = 168$$

3×3 invertible matrices. We refer to the set of $n \times n$ matrices which are safe against distinguishing attacks with advantage ≈ 1 when followed by an application of Toffoli Stream operation as $\mathcal{M}_n$.

We further reduced the search space by enforcing the restriction meant for differences. For 3×3 matrices, this was trivial to achieve. We just had to ensure that difference $(1, 0, 0)$ when multiplied with matrix M should not result in $(1, 0, 0)$. Notice that this only happens when the matrix is in the form of:

$$M = \begin{bmatrix} 1 & \cdot & \cdot \\ 0 & \cdot & \cdot \\ 0 & \cdot & \cdot \end{bmatrix}$$

where the dots are 1 or 0. This matrix always gives the output difference $(1, 0, 0)$, given $(1, 0, 0)$ as the input difference. This meant we had one less column we had to consider for the first column. This reduced the amount of matrices we needed to search to $(2^3 - 1 - 1)(2^3 - 2)(2^3 - 2^2) = 144$. Among these 144 matrices, we needed to find the ones that ensured the restriction on linear masks.

The algorithm that checks this restriction for 6×6 is shown in Algorithm 5 (TRAIL-CHECK). This algorithm counts the number of linear trails where every mask in the trail is in the form of (non-zero, _, _) when a matrix M is applied repeatedly on some output mask.

Algorithm 5 TRAIL-CHECK

```
1: function TRAIL_CHECK(M)
2:     count ← 0
3:     found ← false
4:     // If we are testing a matrix with size n, then
5:     // the size of v⃗ is 2n/3 (I.e., the size of two limbs)
6:     for each bitstring v⃗ except (0, 0, ..., 0, 0) do
7:         r⃗ ← M^T · (0^{n/3} ‖ v⃗)
8:         // If we immediately found that M^T · r⃗ = (non-zero,_,_) then
9:         // we do not need to check whether (M^T)^i · r⃗ = (non-zero,_,_)
10:        // for i > 1.
11:        if first limb of r⃗ ≠ 0^{n/3} then
12:            continue
13:        end if
14:        for ord(M^T) − 2 times do
15:            r⃗ ← M^T · r⃗
16:            if the first limb mask of r⃗ ≠ 0^{n/3} then
17:                found ← true
18:                break from for loop.
19:            end if
20:        end for
21:        if NOT found then
22:            count ← count + 1
23:        end if
24:        found ← false
25:    end for
26:    return count
27: end function
```

Algorithm 5 is then used as an assessment function for a matrix that is tried in Algorithm 6. Algorithm 6 (TRY-MATRICESM3) counts and saves the suitable matrices of size 3 based on the criteria we defined. It turns out that out of 144 candidates derived with the restriction on difference and invertibility criteria, 96 of them are suitable matrices.

Algorithm 6 TRY-MATRICESM3

```
1: // The set of suitable 3 × 3 matrices M_3 is initially empty.
2: function TRY_MATRICESM3(M_3)
3:     safe_count ← 0
4:     for each matrix M ∈ GL(3, 2) do
5:         if det(M) mod 2 > 0 then
6:             lin_amount ← TRAIL_CHECK(M)
7:             if lin_amount = 0 then
8:                 safe_count ← safe_count + 1
9:                 Add(M_3, M)
10:            end if
11:        end if
12:    end for
13:    return (safe_count, M_3)
14: end function
```

For calculating suitable matrices on matrices of size > 3, Algorithm 7 (TRY-MATRICES) is used. This function calculates the number of suitable matrices extended from suitable matrices of lower dimensions.

Algorithm 7 TRY-MATRICES

```
1: function TRY_MATRICES(M_n, M_2n)
2:     safe_count ← 0
3:     for each matrix M in M_n do
4:         // Different combinations of possible matrix expansions.
5:         combinations ← expansions(M)
6:         for each matrix C in combinations do
7:             if det(C) mod 2 > 0 then
8:                 lin_amount ← TRAIL-CHECK(C)
9:                 if lin_amount = 0 then
10:                    safe_count ← safe_count + 1
11:                    Add(M_2n, C)
12:                end if
13:            end if
14:        end for
15:    end for
16:    return (safe_count, M_2n)
17: end function
```

We extended our search to 2-bit repeating patterns deriving from the candidates $\mathcal{M}_3$. The `expansions` operation from a matrix in $\mathcal{M}_3$ to a 6×6 matrix is done as follows: all bits which were 1 become either $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ or $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$. All bits which were 0 become

either $\begin{bmatrix}0 & 0\\0 & 0\end{bmatrix}$ or $\begin{bmatrix}1 & 1\\1 & 1\end{bmatrix}$.

Example:

$$M = \begin{bmatrix}1 & 1 & 0\\0 & 1 & 0\\0 & 0 & 1\end{bmatrix}$$

A valid expansion would be:

$$\begin{bmatrix}\begin{bmatrix}1 & 0\\0 & 1\end{bmatrix} & \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix} & \begin{bmatrix}0 & 0\\0 & 0\end{bmatrix}\\ \begin{bmatrix}1 & 1\\1 & 1\end{bmatrix} & \begin{bmatrix}0 & 1\\1 & 0\end{bmatrix} & \begin{bmatrix}0 & 0\\0 & 0\end{bmatrix}\\ \begin{bmatrix}0 & 0\\0 & 0\end{bmatrix} & \begin{bmatrix}1 & 1\\1 & 1\end{bmatrix} & \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}\end{bmatrix}$$

We found out that all 6×6 matrices deriving from matrices in $\mathcal{M}_3$ are suitable matrices. Since there are two possible square matrices each bit can be expanded to, there are 2^9 different matrix combinations per a matrix from $\mathcal{M}_3$. Since $|\mathcal{M}_3| = 96$, this means $|\mathcal{M}_6| = 2^9 \cdot 96 = 49152$.

Example:

A suitable matrix M where $M \in \mathcal{M}_6$:

$$M = \begin{bmatrix}0 & 0 & 1 & 0 & 0 & 0\\0 & 0 & 0 & 1 & 0 & 0\\0 & 0 & 0 & 0 & 1 & 0\\0 & 0 & 0 & 0 & 0 & 1\\1 & 0 & 0 & 0 & 0 & 0\\0 & 1 & 0 & 0 & 0 & 0\end{bmatrix}$$

$$M^{\mathrm{T}} = \begin{bmatrix}0 & 0 & 0 & 0 & 1 & 0\\0 & 0 & 0 & 0 & 0 & 1\\1 & 0 & 0 & 0 & 0 & 0\\0 & 1 & 0 & 0 & 0 & 0\\0 & 0 & 1 & 0 & 0 & 0\\0 & 0 & 0 & 1 & 0 & 0\end{bmatrix}$$

All of its possible linear trails:

```
[00,10,00] -> [00,00,10] -> [10,00,00] ↺
[00,01,00] -> [00,00,01] -> [01,00,00] ↺
[00,11,00] -> [00,00,11] -> [11,00,00] ↺
[00,10,10] -> [10,00,10] -> [10,10,00] ↺
[00,01,10] -> [10,00,01] -> [01,10,00] ↺
[00,11,10] -> [10,00,11] -> [11,10,00] ↺
```

[00,10,01] –> [01,00,10] –> [10,01,00] ↺
[00,01,01] –> [01,00,01] –> [01,01,00] ↺
[00,11,01] –> [01,00,11] –> [11,01,00] ↺
[00,10,11] –> [11,00,10] –> [10,11,00] ↺
[00,01,11] –> [11,00,01] –> [01,11,00] ↺
[00,11,11] –> [11,00,11] –> [11,11,00] ↺
[01,11,11] –> [11,01,11] –> [11,11,01] ↺
[10,01,11] –> [11,10,01] –> [01,11,10] ↺
[10,10,01] –> [01,10,10] –> [10,01,10] ↺
[10,10,10] ↺
[11,11,11] ↺
[10,11,01] –> [01,10,11] –> [11,01,10] ↺
[01,01,01] ↺
[10,11,10] –> [10,10,11] –> [11,10,10] ↺
[01,01,10] –> [10,01,01] –> [01,10,01] ↺
[11,01,01] –> [01,11,01] –> [01,01,11] ↺
[10,11,11] –> [11,10,11] –> [11,11,10] ↺

The notation $[\alpha, \beta, \gamma]$ is the masks for limbs 1, 2 and 3. As can be seen, there is not a trail where each mask has an all-zero mask in the first limb. (↺ means looping back around to the initial state of the trail.)

As our search space increased by quite a lot, the next dimension expansion was unfeasible to search through so we stopped at dimensions 6×6 for this task. However, with more efficient algorithms and better hardware, we think this search can be expanded, so we would like to mention how we expanded matrices from 6×6 and beyond.

Rule:

Suppose we have a $3n \times 3n$ matrix that is made of 9 circulant block matrices. (If 3×3 matrix, then it is just 9 values instead of blocks.)
For each block, we determine pairs of bitstrings of length n that result in the bitstring that forms the circulant block matrix when we apply the XOR operation on the two.
The concatenation of those pairs to get a bitstring of size $2n$ is the row expansion operation.
After we have such a bitstring, we construct a circulant matrix out of it.
The resulting $2n \times 2n$ square matrix then can be used to replace the old $n \times n$ circulant block matrix.
After doing this for each block, we end up with a new matrix of size $6n \times 6n$.

Example:

$$M = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

This matrix is 3×3, so we look at 1-bit values. For each value x, we check for which pairs of 1-bit values, when an XOR operation is applied, we end up with x. For the cells that are 1 in M, we look for pairs of 1-bit values that result in 1 when added together. These pairs are $(1, 0)$ and $(0, 1)$. We now concatenate the values for each of these pairs

and we end up with the 2-bit vectors $(1, 0)$ and $(0, 1)$. Circulant matrices that can be constructed with these vectors are $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ and $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$. Thus each 1-bit cell with value 1 can be replaced by either of the two matrices.

For cells that have the value 0 in M, we have the pairs $(0, 0)$ and $(1, 1)$. So we have the vectors $(1, 1)$ and $(0, 0)$ and from these vectors, we can construct two circulant matrices: $\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$ and $\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$. Thus each 1-bit cell with value 0 can be replaced by either of the two matrices.

This is the derivation of the expansion rule from $\mathcal{M}_3$ to 6×6 matrices we defined in Page 46.

As dimensions get bigger, the number of different expansion candidates increases exponentially; for this research, we decided to stop expanding the matrices at 12×12. After stopping at 6×6 matrices, we focused on measuring the quality of good matrices we found thus far. Doing so allowed us to gain insight into the security of the matrices when paired with the bitwise AND operation. The measurements were done for 24 rounds of permutation, as this was the default recommended number of permutations specified in [14].

Chapter 8

Measuring the Security of the Permutation with a Chosen Matrix

When using any matrix M from $\mathcal{M}_n$ to construct a permutation like shown in Figure 7.1, the resulting permutation is guaranteed to not be distinguishable with an advantage close to 1 in the case of repeated applications. However, this does not imply that distinguishing attacks are unfeasible. It could be the case that, although the advantage is not approximately 1, it could still be high. This is why next we measured the likelihood of distinguishing a permutation made up of one of the matrices from $\mathcal{M}_n$ over 24 rounds for each matrix and chose the most promising ones.

The DP of any differential trail and the LP of any linear trail must be low. The reason for this is, assuming distinguishing attacks through differential and linear cryptanalysis, the advantage of distinguishing is lower when the DP of a differential trail is lower. The same can be said about the LP of a linear trail. To have clearer measurements, we used weights instead of probabilities, as they provided clearer measurements when probabilities were very small. A weight is calculated as $-\log_2(p)$ where p is the probability.

What we aimed to measure was the weight of the trail with the lowest weight for 24 rounds, given a matrix and a bitwise AND operation as shown in Figure 7.1. It can be inferred from Sections 2.3.1 and 2.3.2 that, constructing a trail from any linear operation is deterministic. However, we also have a non-linear operation, that gives rise to probabilistic outcomes. The effects of the bitwise AND operation are different for differences and linear masks.

Figure 8.1 shows that the input difference (α, β) can propagate to a total of $2^{wt(\alpha \vee \beta)}$ possible differences. The probability of the output difference being a specific one out of them is $2^{-wt(\alpha \vee \beta)}$. Thus, the weight is $wt(\alpha \vee \beta)$ [14].

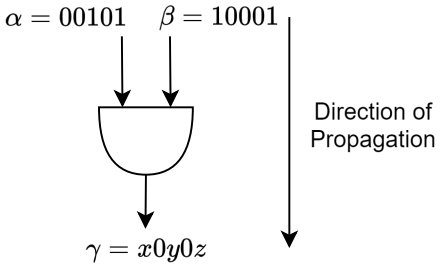


Figure 8.1: bitwise AND on differences

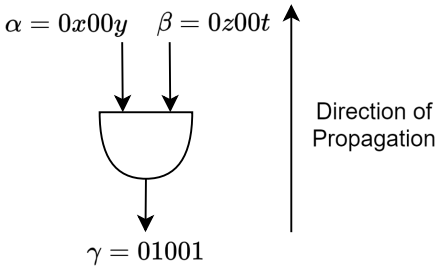


Figure 8.2: bitwise AND on linear masks

Figure 8.2 shows that given an output mask γ the number of possible combinations the pair of input masks (α, β) can take is $4^{wt(\gamma)}$ (We are working with LP values, we get this result from $2^{wt(\gamma)} \cdot 2^{wt(\gamma)} = 2^{2wt(\gamma)} = 4^{wt(\gamma)}$). The probability of the input mask pair being a specific pair out of the possible pairs is $4^{-wt(\gamma)}$. Thus, the weight is $2wt(\gamma)$ [14]

We can represent the relation between states as transitions from one to another with a certain probability. The mathematical model we used to represent this relation was a directed graph. The vertices are differences/linear masks, the edges connecting two states represent transitions and the weight of these edges correspond to the probability of the transition. Graphs with finitely many vertices where the weights of outgoing edges denote probabilities are a representation of a random process called a Markov Chain [13, 13]. These weights add up to 1. In a Markov Chain, the probability of transitioning to the next state depends only on the current state . A path of length n is a sequence of n states that are traversed in the graph.

The transition conditions are different for differential and linear propagation. For differential propagation, given the linear round M and an initial state $\vec{s}$, the possible states that can be transitioned from $\vec{s}$ are computed with the function $f = (\wedge)_{\texttt{diff}} \circ h$.

Where $h(\vec{v}) = M \cdot \vec{v}$ and $(\wedge)_{\texttt{diff}}$ is defined as follows:

$(\wedge)_{\texttt{diff}} : \mathbb{F}_2^{3\ell} \to \mathcal{P}(\mathbb{F}_2^{3\ell})$ where ℓ is a power of 2
$(\wedge)_{\texttt{diff}}(\alpha, \beta, \gamma) = \{(\alpha', \beta, \gamma) \mid$ For every b_i at position i, if $b_i = 0$ then $\alpha'_i = \alpha_i$ otherwise $\alpha'_i \in \{0, 1\}\}$

where: $\beta \vee \gamma = b_0 b_1 \dots b_{\ell-1}$,
$x_i :=$ bit value of limb x at position i (also zero-indexed)
ℓ is the length of the bit pattern of each limb.

The function f results in $2^{wt(\beta' \vee \gamma')}$ different states from an intermediate state $s' = (\alpha', \beta', \gamma')$ where s' is the result of the matrix multiplication.

Example: Suppose we have an initial state $s = (1, 0, 0)$ and a matrix

$$M = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

and we want to compute the graph (for one transition) for differential propagation. Then, the graph looks as in Figure 8.3.

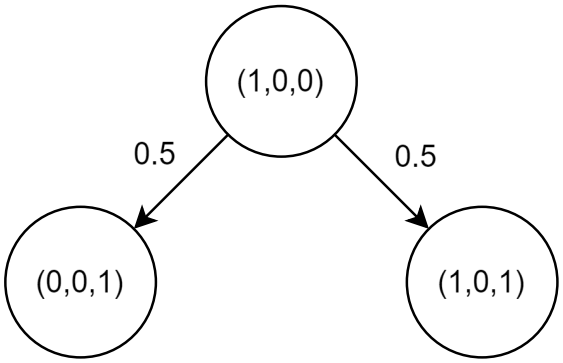


Figure 8.3: Transition graph for differential propagation

First, we multiply s with the matrix M and get

$$s' = M \cdot s = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

Then, we apply the bitwise AND operation on the intermediate state. I.e., bitwise AND operation on limbs 2 and 3. The result becomes the difference x where $x \in \{0, 1\}$. Then, the XOR operation between limb 1 difference and x is applied, which results in limb $1 = x \oplus$ limb 1.

In the generalized case, for patterns of bits which has a length greater than 1, the first limb would take $4^{wt(\alpha')}$ values, thus, we would need to consider $4^{wt(\alpha')}$ states to transition. The last two limbs are left unchanged as depicted in Figure 7.1. Since $wt(0 \vee 1) = 1$, the weight of each edge is $2^{-1} = 0.5$. If we write down all the different combinations of $(x, 0, 1)$, we get $\{(0, 0, 1), (1, 0, 1)\}$. Hence, the graph in Figure 8.3.

The full graph is given in Figure 8.4

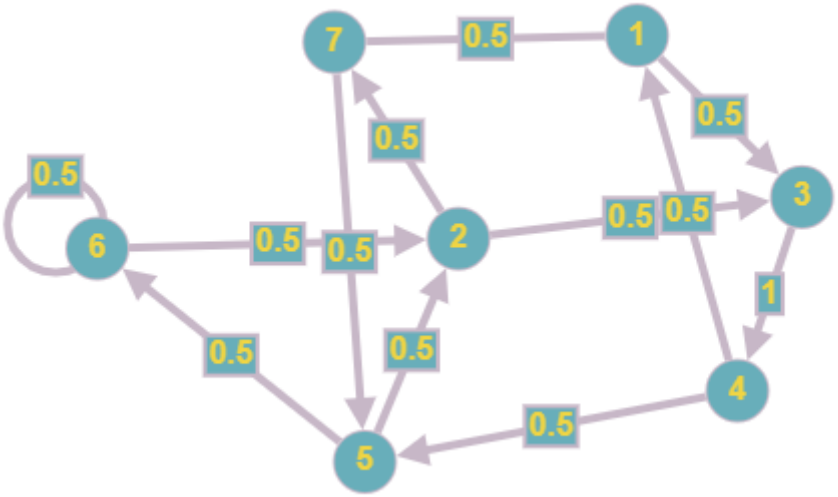


Figure 8.4: Full transition graph for differential propagation

In Figure 8.4, the vertices are natural numbers. Each natural number represents the state of its binary representation. For example, $1 = \mathbf{001}$, $4 = \mathbf{100}$ and $5 = \mathbf{101}$. As can be seen, two edges are going outwards from 4 to 1 and 5 with a probability of 0.5, just like we saw earlier.

For linear propagation, given the linear round M and an initial state $\vec{s}$, $f' = g \circ (\wedge)_{\texttt{lin}}$ is used to compute the states that can be transitioned to from $\vec{s}$. Where $g(X) = \{t(\vec{x}) \mid \vec{x} \in X\}$, $t(\vec{v}) = M^{\mathrm{T}} \cdot \vec{v}$ and $(\wedge)_{\texttt{lin}}$ is defined as follows:

$$
\begin{aligned}
&(\wedge)_{\texttt{lin}} : \mathbb{F}_2^{3\ell} \to \mathcal{P}(\mathbb{F}_2^{3\ell}) \\
&(\wedge)_{\texttt{lin}}(\alpha, \beta, \gamma) = \{(\alpha, \beta', \gamma') \mid \text{if } \alpha_i = 0 \text{ then } \beta'_i = \beta_i \text{ and } \gamma'_i = \gamma_i \\
&\text{otherwise } \beta'_i, \gamma'_i \in \{0, 1\}\}
\end{aligned}
$$

In this case, the bitwise AND operation is applied first. I.e., the (transposed) matrix multiplication is applied to a set of outputs instead of a single state. In this case, the result of the $(\wedge)_{\texttt{lin}}$ on vector $\vec{v} = (\alpha, \beta, \gamma)$ results in $4^{wt(\alpha)}$ different vectors representing a linear mask.

Example: Take the same matrix M from the previous example, then:

$$M^{\mathrm{T}} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

$(\wedge)_{\texttt{lin}}$ is applied first.

$(\wedge)_{\texttt{lin}}(1,0,0) = \{(1,0,0),(1,1,0),(1,0,1),(1,1,1)\}$. If we then multiply the results with M^{T}, we get:

$$\{(1,0,1),(1,1,0),(0,1,0),(0,0,1)\}$$

Each transition has a probability of $4^{-wt(1)} = 4^{-1} = 0.25$ occurring. Thus, one round of transition from state $(1,0,0)$, i.e., from state 4 is graphed as in Figure 8.5.

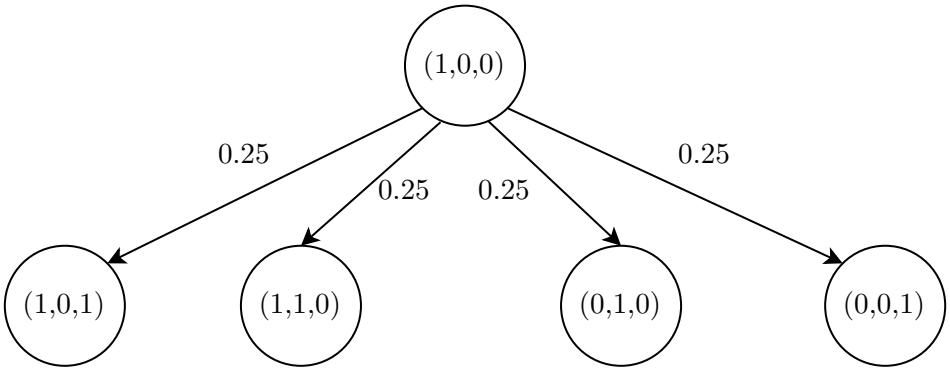


Figure 8.5: Transition graph for linear propagation

Similarly, the full linear transition graph is given in Figure 8.6:

If we apply this technique to generate transitions for every state, we would generate the Markov Chain for the entire state space. The probability of transitioning to a state s' in the nth transition given that we start from state s is denoted as $\Pr(X_{24} = s' \mid X_0 = s)$. As we have discussed, if there is a path for which the probability is high both for the differential and linear case, then this would imply a high advantage against distinguishing attacks. So the quality of a linear round is based on how high the highest probability path is. A naive approach to this problem would be, assuming 24 rounds of permutations are done, we would be looking for $\max\limits_{s,s'} \Pr(X_{24} = s' \mid X_0 = s)$, the highest probability starting from some state s and ending up at s' in the 24th transition. One way to calculate this probability is to find the cell with the biggest value in C^{24} where C is the transition matrix. This method works fine in (approximately) $\mathcal{O}(n^3)$ since matrix multiplication is upper-bounded around this value, however, this approach does not exactly answer our problem. Evaluating the value of $\Pr(X_{24} = s' \mid X_0 = s)$ calculates the probability of starting at state s and ending at state s'. However, this value is not equivalent to the probability of a singular path. Instead, this value is equivalent to the sum of probabilities of paths starting from state s and ending at s'. We can derive this

Figure 8.6: Full transition graph for linear propagation

as shown in Equation (8.1) from the definition of conditional probability and the law of total probability [19, 30–31].

$$\Pr(X_{24} = s' \mid X_0 = s) = \\ \frac{1}{\Pr(X_0 = s)} \sum_{s_1,\ldots,s_{23}} \Pr(X_{24} = s', X_0 = s, X_1 = s_1, \ldots, X_{23} = s_{23}) \tag{8.1}$$

Hence, we needed to change our approach to focus on individual paths and finding the path with the highest probability instead of just determining the highest probability of ending up in some state s' in n steps starting from another state s.

We want to identify the path of length 24 that has the highest probability (lowest weight). The probability of the path is calculated as:

$$\prod_{(s,s')\in T} w(s, s')$$

where T is the sequence of transitions taken and $w(s, s')$ is the weight of an edge/-transition. While looking for inspiration on how we could write such an algorithm, we found that there is a graph algorithm that solves a similar problem. We took inspiration from https://www.geeksforgeeks.org/longest-path-in-a-directed-acyclic-graph-dynamic-programming/, which is an algorithm that finds the longest path in a directed

acyclic graph. This can easily be modified to find the path with the highest weight. However, this algorithm does not work when the graph contains loops, whereas in our case, a transition can happen to a previously visited state. We modified this algorithm to limit the length of the path so that we have a new stopping condition and we do not check whether a state has been visited before. This small but important change allowed us to visit the same vertices multiple times since we now have an upper bound on how many times we take a step in the graph. Similar to the algorithm, our solution will also be based on dynamic programming. After doing additional research, we found out that this modified algorithm is somewhat of a variant of an already existing algorithm called the **Viterbi Algorithm** [16]. We made a variant of the algorithm that deals with Markov chains instead of hidden Markov models. I.e., a variant without any observations where the state of the graph and its probability distribution for every transition between each state is known.
The algorithm we implemented uses the following recursive relation to compute the highest probability path probability:

$$\begin{aligned} &dp[s, 0] = 1 \\ &dp[s, limit] = \max_{child \in s.children} \{w(s, child) \cdot dp[child, limit - 1]\} \end{aligned}$$

In our case, the limit value was set to 24. So our algorithm finds the value where: $dp[s_i, 24] = \max_{s', s_{23}, \ldots, s_1} \Pr(X_{24} = s', X_{23} = s_{23}, \ldots, X_1 = s_1 \,|\, X_0 = s_i)$. This dynamic programming solution allows us to backtrack and save the path we take:

$$path[s, 0] = []$$

$$path[s, limit] = [s'] \;++\; path[s', limit - 1]$$

where "++" is the concatination operation and s' is equal to s_1 in:

$$\operatorname*{argmax}_{s_{limit-1}, \ldots, s_1} \Pr(X_{limit-1} = s_{limit-1}, X_{limit-2} = s_{limit-2}, \ldots, X_1 = s_1 \,|\, X_0 = s)$$

After we calculated the matrices $\mathcal{M}_3$, we ran the algorithm on these matrices. After running this algorithm, we discovered that all of these matrices have the same weight distributions. We stored these distributions in dictionaries. Dictionaries are a type of collection that associates keys with values. The dictionary keys are the weights of the path with the highest probability in n rounds and the values of the dictionary are the number of states that have such a weight starting from that state. Dict(lowest weight => number of states)

$$\begin{aligned} \text{diff weights: } &\text{Dict}(16.0 => 7) \\ \text{lin weights: } &\text{Dict}(16.0 => 7) \end{aligned}$$

For 24 rounds, all seven states (excluding the all-zero state) that we can form with three bits have a weight of 16. Hence, we could not infer enough information from

3×3 matrices. Therefore, we ran the same algorithm for 6×6 matrices to look for distinguishing factors. For every subset of $\mathcal{M}_6$ calculated from a $\mathcal{M}_3$ matrix, (i.e., 2^9 configurations for each $\mathcal{M}_3$) we created a file to analyze their qualities with the same algorithm. After running the algorithm on all unique matrices (by unique, we mean excluding matrices which are equal to another matrix's transpose), we saw that there were different varieties of weight values and distributions. We saw that in some matrices, the lowest linear weights had considerably bigger values (28.0) but conversely, had smaller lowest differential weights (23.0). The same goes for the other way around. We saw 26.0 for the differential case and 22.0 for the linear case. We deemed the more balanced and high lowest weights to be good candidates for a good matrix. So we picked the matrices with the lowest weights 26.0 and 26.0.

We then looked at the distribution of weights and further narrowed down the matrices with the lowest weights appearing less. There were mainly two different weight distributions.

First distribution:

$$\begin{aligned} &\text{diff weights: Dict}(27.0 => 39, 26.0 => 4, 28.0 => 20) \\ &\text{lin weights: Dict}(26.0 => 19, 30.0 => 16, 28.0 => 28) \end{aligned} \tag{8.2}$$

Second distribution:

$$\begin{aligned} &\text{diff weights: Dict}(29.0 => 8, 27.0 => 20, 26.0 => 2, 28.0 => 33) \\ &\text{lin weights: Dict}(26.0 => 25, 28.0 => 38) \end{aligned} \tag{8.3}$$

Observing the differences between Distributions 8.2 and 8.3, we can see that in the first distribution, the lowest linear weight appears for 19 states, whereas for the second distribution, there are 25. Conversely, in the first distribution, the lowest differential weight appears in four states, whereas in the second distribution, there are two.

However, notice that the first distribution has 26.0 appearing in six states less in linear weights, and the second distribution has 26.0 appearing in two states less in differential weights. Moreover, in the second distribution, the highest differential weight (29.0) is higher than that of the first distribution but the highest differential weight value in the two distributions only differs by one, and eight of the states have this highest weight value. However, the first distribution has the highest linear weight value (30.0), and the highest linear weights between the two distributions differ by two instead of one. Since the weights are on a logarithmic scale to probabilities, a weight difference of n is 2^{-n} factor of difference in probability. Thus, even small differences in weights represent significant differences in probability. Additionally, 16 states have the highest linear weight in the first distribution compared to 8 states having the highest differential weight value in the second distribution. Because of these reasons, we prioritized matrices with the first distribution when doing further computations, since except for small disadvantages in differential weights compared to the second distribution, the first distribution is more favorable due to bigger advantages in linear weights.

Finally, we also filtered the matrices that had less than 14 1s in them to be more cost-effective, as a high number of 1s could make the software implementation costly.

With this, we ended up with 8 matrices of cost 10 and 30 matrices of cost 12 for the next analysis.

On a side note, we would like to highlight an interesting property we noticed. We visualized the transition graph for one of the matrices. The matrix is:

$$M = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

The transition graph for differential propagation is depicted in Figure 8.7.

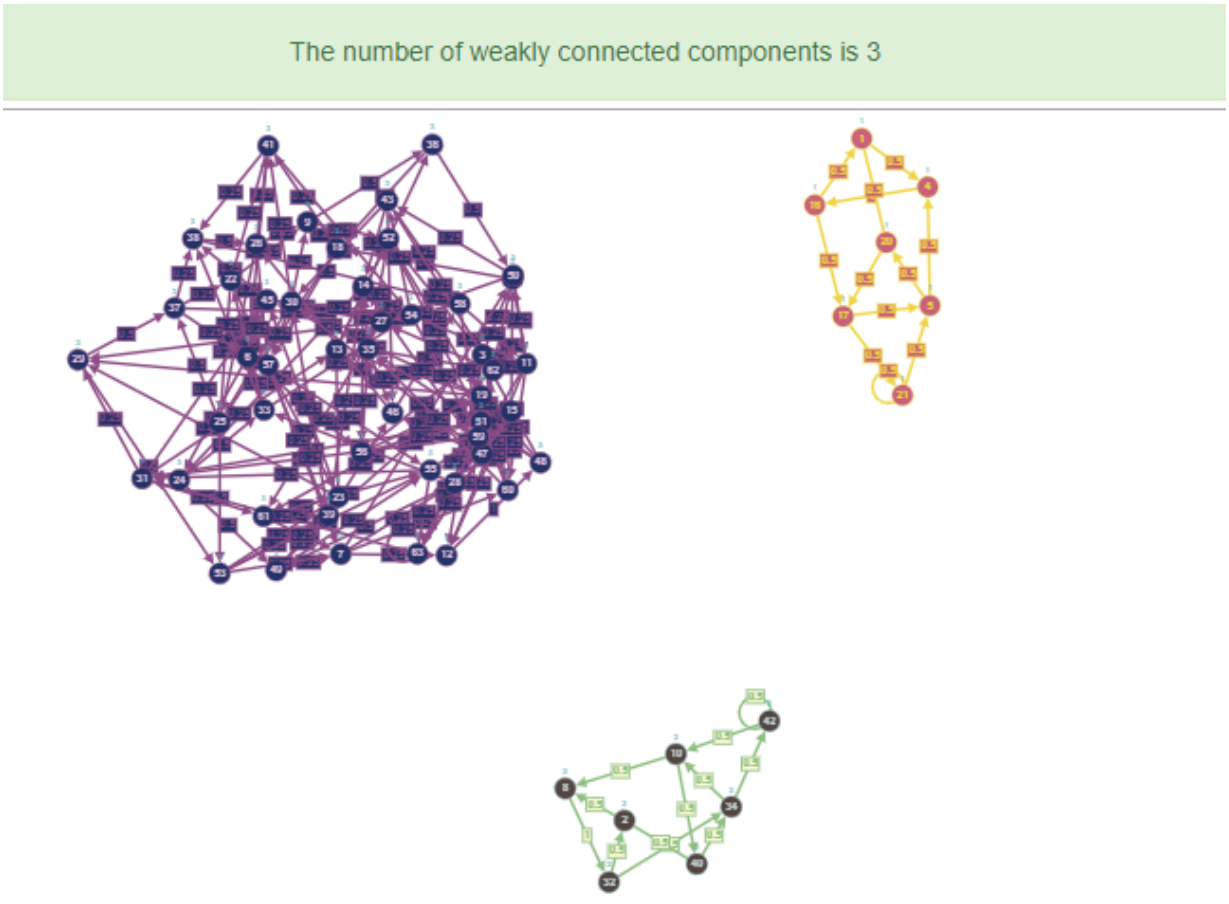


Figure 8.7: Transition graph with multiple connected components

Since the graph is a bit clustered, it is hard to see individual transitions, however, the interesting thing to notice is that this graph has multiple connected components. Other

graphs we looked at only have one connected component. The existence of multiple connected components implies that, for certain starting differences, it is impossible to transition to some differences. We believe the reason for this is that the columns in odd positions only affect the bit positions at odd positions and the columns in even positions affect the bit positions at even positions. This means that only a subset of modifications to a state is possible before we eventually loop back to one of the previous modifications. The matrix itself is also the identity matrix but shifted 2 bits to the right.

Chapter 9

Algebraic Degree Analysis

The final analysis we did on the matrices was analyzing the algebraic degree of the permutation comprised of the matrix and the Toffoli Stream operation. The permutation must have a high algebraic degree for the reasoning outlined in [14]. Given a variable $x = a\|b\|c$ where a, b and c are variables in $\mathbb{F}_2^\ell$, we can think of a permutation F as a vector boolean function from $\mathbb{F}_2^{3\ell}$ to $\mathbb{F}_2^{3\ell}$ where ℓ is the length of the repeating bit pattern in each limb and M is the matrix we are analyzing.

As outlined in Figure 7.1, we define the permutation F_M as $F_M = \text{Toffoli} \circ f_M$, where $f_M(a, b, c) = M \cdot (a, b, c)$ and $\text{Toffoli}(a, b, c) = (a + bc,\ b,\ c)$ (Toffoli Stream operation).

After we narrowed down the matrices to 38 matrices that follow one of two distributions and have and divided them based on Distributions 8.2 and 8.3, we started our analysis on matrices from the first distribution for the reasons we described. For the analysis, we wanted to measure the degree of up to 6 rounds for each of the matrices we picked. However, instead of measuring the degree for 128-bit long repeating bit patterns for 6 rounds, we decided to measure the degree of the permutation starting from 1-bit long bit patterns up to the longest bit pattern we could test in a reasonable amount of time for 6 rounds; the longest bit patterns we tested were 4-bit long. We decided to reduce the length of bit patterns of the limbs to decrease computational and memory requirements, similar to what we did in previous analyses. While this reduction impacts the accuracy of the degree analysis by imposing an upper bound of 3ℓ on the maximum achievable degree, we are still able to observe how the degree increases for each round up to 6 rounds. This approach enables us to estimate whether the permutation constructed with M will have a high algebraic degree over 6 rounds with a longer bit pattern based on the fact that it shows a promising growth in the degree with a short pattern length. The pseudocode for the algorithm is given in Algorithm 8.

Here, $\dim(M)$ returns the dimensions of the square matrix M and $\deg(\vec{v})$ returns the degree of the vector $\vec{v}$. In the software implementation, we recorded the degree of each limb separately, however, just focusing on the degree of the first limb is enough to determine the degree of the permutation. I.e., to determine the degree of the boolean vector function $\vec{v} = (a, b, c)$, where
$\vec{v} \in \{f \mid f : \mathbb{F}_2^{3\ell} \to \mathbb{F}_2^{3\ell}\}$ and a, b, $c \in \{f \mid f : \mathbb{F}_2^{3\ell} \to \mathbb{F}_2^{\ell}\}$ and ℓ is the length of the

Algorithm 8 ALGEBRAIC-DEGREE

```
1: function ALGEBRAIC_DEGREE(degrees, M)
2:     size ← dim(M)
3:     // initializing a size × 1 vector made up of boolean variables.
4:     v⃗ ← (x_1, x_2, ..., x_size)
5:     for 6 times do
6:         v⃗ ← F_M(v⃗)
7:         Add(degrees, deg(v⃗))
8:     end for
9:     return degrees
10: end function
```

repeating bit pattern for each limb, the following formula can be used:

$$\deg(\vec{v}) = \deg(a)$$

Another thing we did was to introduce an early stopping condition. As we discussed, although we can not determine the algebraic degree of the function for 384 bits directly, we could estimate whether it would be large; we stopped considering the matrix if, by the second application of the permutation, the degree is not 4. This is because, in an ideal scenario, the degree is expected to double per permutation. If the degree growth initially follows more of a linear growth than an exponential growth, then we can assume by the sixth round, it will have a small degree.

Example: Take matrix M where

$$M = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \end{bmatrix}$$

The first two rounds of permutation F_M are as follows

```
Initial vector:
    [x1,x2,x3,x4,x5,x6]
Round 1:
Vector: [x1*x2 + x1*x3 + x1*x6 + x2*x4 + x2 + x3*x4 + x4*x6, x1*x2
        + x1*x3 + x1 + x2*x4 + x2*x5 + x3*x4 + x3*x5, x2 + x3 + x6,
        x1 + x4 + x5, x1 + x4, x2 + x3]
Limb 1: [2, 2]
Limb 2: [1, 1]
Limb 3: [1, 1]
```

```
Round 2:
Vector: [x1*x2*x3 + x1*x2*x4 + x1*x2*x5*x6 + x1*x2*x5 + x1*x3*x4
        + x1*x3*x5*x6 + x1*x3*x5 + x1*x4*x6 + x1*x4 + x1*x5 + x1*x6
        + x2*x3*x4 + x2*x3*x5 + x2*x4*x5*x6 + x2*x4*x5 + x2*x5 + x2*x6
        + x3*x4*x5*x6 + x3*x4*x5 + x3*x4 + x5*x6, x1*x2*x3 + x1*x2*x4
        + x1*x2*x5*x6 + x1*x2*x6 + x1*x3*x4 + x1*x3*x5*x6 + x1*x3*x6
        + x1*x4*x6 + x1*x5 + x1*x6 + x2*x3*x4 + x2*x3*x5 + x2*x3
        + x2*x4*x5*x6 + x2*x4*x6 + x2*x5 + x2*x6 + x3*x4*x5*x6
        + x3*x4*x6 + x3*x4 + x5*x6, x1*x2 + x1*x3 + x1 + x2*x4 + x2*x5
        + x3*x4 + x3*x5 + x6, x1*x2 + x1*x3 + x1*x6 + x2*x4 + x2
        + x3*x4 + x4*x6 + x5, x1*x2 + x1*x3 + x1*x6 + x1 + x2*x4 + x2
        + x3*x4 + x4*x6 + x4 + x5, x1*x2 + x1*x3 + x1 + x2*x4
        + x2*x5 + x2 + x3*x4 + x3*x5 + x3 + x6]
Limb 1: [4, 4]
Limb 2: [2, 2]
Limb 3: [2, 2]
```

In this example, the expression Limb i represents a list where each entry indicates the degree of the Boolean function that describes each bit in the ith limb. Specifically, the jth entry in Limb i denotes the degree of the Boolean function for the jth bit of the ith limb. The degree of the permutation is the maximum of the degrees of the limbs. The degree of the permutation F_M in the 3rd, 4th, 5th, and 6th rounds are 4, 5, 5, and 5.

An interesting discovery we made for 6×6 matrices was the following relation:

$$\text{From second distribution} \implies \text{Low algebraic degree} \tag{9.1}$$

Every matrix we tried from the second distribution showed linear growth instead of exponential growth. We are not sure as to why this may be the case since we did not analyze the mathematical properties of these matrices. We suspect that it may have to do with the number of 1s or less likely, the configuration of these matrices. The matrices from the second distribution that were filtered out are as follows:

$$\begin{bmatrix} . & 1 & . & . & . & . \\ 1 & . & . & . & . & . \\ . & . & 1 & . & . & 1 \\ . & . & . & 1 & 1 & . \\ 1 & . & . & 1 & . & . \\ . & 1 & 1 & . & . & . \end{bmatrix}$$

$$\begin{bmatrix} 1 & . & . & . & . & . \\ . & 1 & . & . & . & . \\ . & . & . & 1 & 1 & . \\ . & . & 1 & . & . & 1 \\ . & 1 & 1 & . & . & . \\ 1 & . & . & 1 & . & . \end{bmatrix}$$

$$\begin{bmatrix} . & 1 & . & . & . & . \\ 1 & . & . & . & . & . \\ 1 & . & . & . & . & 1 \\ . & 1 & . & . & 1 & . \\ . & . & . & 1 & 1 & . \\ . & . & 1 & . & . & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & . & . & . & . & . \\ . & 1 & . & . & . & . \\ . & 1 & . & . & 1 & . \\ 1 & . & . & . & . & 1 \\ . & . & 1 & . & . & 1 \\ . & . & . & 1 & 1 & . \end{bmatrix}$$

$$\begin{bmatrix} . & 1 & . & . & . & . \\ 1 & . & . & . & . & . \\ . & . & 1 & . & . & 1 \\ . & . & . & 1 & 1 & . \\ 1 & . & . & 1 & . & . \\ . & 1 & 1 & . & . & . \end{bmatrix}$$

$$\begin{bmatrix} 1 & . & . & . & . & . \\ . & 1 & . & . & . & . \\ . & . & . & 1 & 1 & . \\ . & . & 1 & . & . & 1 \\ . & 1 & 1 & . & . & . \\ 1 & . & . & 1 & . & . \end{bmatrix}$$

$$\begin{bmatrix} . & 1 & . & . & . & . \\ 1 & . & . & . & . & . \\ 1 & . & . & . & . & 1 \\ . & 1 & . & . & 1 & . \\ . & . & . & 1 & 1 & . \\ . & . & 1 & . & . & 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 & . & . & . & . & . \\ . & 1 & . & . & . & . \\ . & 1 & . & . & 1 & . \\ 1 & . & . & . & . & 1 \\ . & . & 1 & . & . & 1 \\ . & . & . & 1 & 1 & . \end{bmatrix}$$

All matrices from both distributions by the 6th round, had a degree of 5. The distinguishing factor was the rate of growth. This allowed us to filter out the second distribution and filter out a few matrices from the first distribution. After the filter, we had 22 matrices left.

We then expanded the 6×6 matrices that show promising degree growth to 12×12 matrices for the algebraic degree analysis. To decide which matrices to prioritize when expanding, we determined which matrices have the highest degree in the 6th round and the majority of rounds as a tiebreaker. Multiple matrices can have the same degree by the end of the 6th round, so to determine which high-degree matrices to use first, we looked at the degrees of the 5th and potentially the 4th rounds and prioritized the ones with the higher degrees.

Since there were many ways we could expand a matrix from 6×6 dimensions and higher, we decided to use the Hamming weight as a heuristic. We mentioned in Rule 7.1 how we expand matrices with circulant block submatrices. In the step where we choose a pair of vectors that add up to the vector pattern that forms the circulant block matrix, instead of going through all possible pairs of vectors, we chose the vectors $\mathbf{0}^\ell$ and c where c is the vector pattern that forms the circulant block matrix and ℓ is the length of the bit pattern of each limb. This is a valid pair since $\mathbf{0}^\ell \oplus c = c$, thus we can expand the circulant submatrix with another one where the circulant matrix is generated by the pattern $c \| 0^\ell$.

The 12×12 matrices we analyzed from the 6×6 matrices with promising growth all showed exponential growth. Eighteen of these matrices ended up with degree 10 and four of these matrices ended up with degree 11.

Here are the matrices with degree 11 after 6 rounds:

$$M_1 = \begin{bmatrix}
. & 1 & . & . & . & . & . & . & . & . & . & . \\
. & . & 1 & . & . & . & . & . & . & . & . & . \\
. & . & . & 1 & . & . & . & . & . & . & . & . \\
1 & . & . & . & . & . & . & . & . & . & . & . \\
. & 1 & . & . & 1 & . & . & . & . & 1 & . & . \\
. & . & 1 & . & . & 1 & . & . & . & . & 1 & . \\
. & . & . & 1 & . & . & 1 & . & . & . & . & 1 \\
1 & . & . & . & . & . & . & 1 & 1 & . & . & . \\
1 & . & . & . & . & 1 & . & . & . & . & . & . \\
. & 1 & . & . & . & . & 1 & . & . & . & . & . \\
. & . & 1 & . & . & . & . & 1 & . & . & . & . \\
. & . & . & 1 & 1 & . & . & . & . & . & . & .
\end{bmatrix}$$

$$M_2 = \begin{bmatrix}
1 & . & . & . & . & . & . & . & . & . & . & . \\
. & 1 & . & . & . & . & . & . & . & . & . & . \\
. & . & 1 & . & . & . & . & . & . & . & . & . \\
. & . & . & 1 & . & . & . & . & . & . & . & . \\
1 & . & . & . & . & 1 & . & . & 1 & . & . & . \\
. & 1 & . & . & . & . & 1 & . & . & 1 & . & . \\
. & . & 1 & . & . & . & . & 1 & . & . & 1 & . \\
. & . & . & 1 & 1 & . & . & . & . & . & . & 1 \\
. & 1 & . & . & 1 & . & . & . & . & . & . & . \\
. & . & 1 & . & . & 1 & . & . & . & . & . & . \\
. & . & . & 1 & . & . & 1 & . & . & . & . & . \\
1 & . & . & . & . & . & . & 1 & . & . & . & .
\end{bmatrix}$$

$$M_3 = \begin{bmatrix}
. & 1 & . & . & . & . & . & . & . & . & . & . \\
. & . & 1 & . & . & . & . & . & . & . & . & . \\
. & . & . & 1 & . & . & . & . & . & . & . & . \\
1 & . & . & . & . & . & . & . & . & . & . & . \\
1 & . & . & . & . & . & . & . & . & 1 & . & . \\
. & 1 & . & . & . & . & . & . & . & . & 1 & . \\
. & . & 1 & . & . & . & . & . & . & . & . & 1 \\
. & . & . & 1 & . & . & . & . & 1 & . & . & . \\
. & 1 & . & . & . & 1 & . & . & 1 & . & . & . \\
. & . & 1 & . & . & . & 1 & . & . & 1 & . & . \\
. & . & . & 1 & . & . & . & 1 & . & . & 1 & . \\
1 & . & . & . & 1 & . & . & . & . & . & . & 1
\end{bmatrix}$$

$$M_4 = \begin{bmatrix}
1 & . & . & . & . & . & . & . & . & . & . & . \\
. & 1 & . & . & . & . & . & . & . & . & . & . \\
. & . & 1 & . & . & . & . & . & . & . & . & . \\
. & . & . & 1 & . & . & . & . & . & . & . & . \\
. & 1 & . & . & . & . & . & . & 1 & . & . & . \\
. & . & 1 & . & . & . & . & . & . & 1 & . & . \\
. & . & . & 1 & . & . & . & . & . & . & 1 & . \\
1 & . & . & . & . & . & . & . & . & . & . & 1 \\
1 & . & . & . & 1 & . & . & . & . & 1 & . & . \\
. & 1 & . & . & . & 1 & . & . & . & . & 1 & . \\
. & . & 1 & . & . & . & 1 & . & . & . & . & 1 \\
. & . & . & 1 & . & . & . & 1 & 1 & . & . & .
\end{bmatrix}$$

Where "." symbols in the matrices are 0. A matrix having a degree of 11 does not necessarily mean it will have a higher degree when expanded to longer bit patterns than matrices that have a degree of 10. It could be the case that some monomials canceled each other out and for longer patterns, the matrix with degree 10 could have a higher degree for longer bit patterns. However, picking the highest degree is a good (greedy) heuristic when deciding which matrices to expand.

In the end, we stopped our analysis with four 12×12 matrices that resulted in an algebraic degree of 11 and eighteen 12×12 matrices with an algebraic degree of 10 that need more analysis to choose which of them to expand further.

Chapter 10

Conclusions

In this paper, we outlined a methodology for finding a fix for Friet. We first reduced the problem to fixing the primitive, as it was mentioned in [17]. We initially tried to fix Friet-PC by doing small modifications to its linear layer. We tested for multiple-round differentials and linear approximations where the DP of these differentials and the LP of these linear approximations could equal to 1 within 24 rounds. We found that although the proposals did not contain any multi-round differentials with DP equal to 1, there were still multi-round linear approximations with LP equal to 1 within 24 rounds. We then focused on constructing a linear layer from scratch. To do this, we interpreted the linear section as a 384×384 square matrix. Due to the size of the input, we started constructing matrices to analyze for repeating patterns of bits. Initially, we started with one-bit long patterns for each limb. This meant analyzing 3×3 invertible matrices. We proposed that a matrix M would be up for consideration iff a permutation made from M first ensured that the DP of all differential trails and the LP of all linear trails are less than 1 for r rounds where $r = \text{ord}(r)$. We conducted this for 3×3 matrices and 6×6 matrices that are derived from suitable 3×3 matrices. We then conducted trail analyses on permutations made from these matrices to look for the lowest-weight trails and compare these weights. After filtering the matrices with promising weights and weight distributions, we then conducted algebraic degree analysis on the permutations made from these matrices and expanded the dimensions up to 12×12.

By the end of this research, although we could not find any fixes, we managed to narrow down the search space by expanding from suitable matrices to other suitable matrices and disregarding any other matrix that is not a suitable matrix when reduced down to lower dimensions. We believe that considering the total number of different invertible matrices even at 6×6 dimensions ($\approx 2.015 \cdot 10^{10}$), we narrowed down the search space by quite a lot ($2^9 \cdot 96 = 49152$). Continuing our search from the bottom up as we have done thus far could prove to be truly efficient since we are expanding in a tiny portion of the entire search space. With a more automated decision process, we could analyze more matrices in a shorter amount of time.

Additionally, we found that for permutations constructed from some matrices, there exists a set of differentials (Δ, Δ') such that $\text{DP}(\Delta, \Delta') = 0$ for i-rounds for <u>all</u> $i > 0$,

as shown briefly in Figure 8.7. We highlighted that there is a direct correlation between the distribution of linear & differential weights and the algebraic degree growth rate as mentioned in Implication 9.1. Finally, although this is not concrete as we could not go through each distribution for each matrix, we wanted to mention this. We noticed a relation between the differential & linear weight distributions over r rounds and their lowest weight trail weight; suppose we are given a permutation with a unique (lowest-weight differential trail weight, lowest-weight differential trail) = (x, y) pair after 24 rounds. Then its differential weight distribution and linear weight distribution pair can only be from a fixed set of distributions regardless of which matrix we expanded from. In other words, if we refer back to Distributions (8.2) and (8.3) we notice that any permutation which exhibited the lowest linear weight and the lowest differential weight of 26.0 fell into two categories and no other distributions. This was also the case for matrices with 20 1s, so this phenomenon is not limited to matrices with costs 10 and 12. However, this last finding was discovered recently and does not have as strong enough of a ground to stand on as the other findings we discussed.

Chapter 11

Future Work

Although we did not determine a specific 384×384 matrix, we can acquire one by expanding one of the suitable matrices that have good linear and differential weight distributions (like the Distributions 8.2 and 8.3) and a good algebraic degree. However, with the current methodology we outlined in this paper, conducting a trail analysis on 384-bit patterns is unfeasible.
Currently, the main problem we face with analyzing matrices as dimensions grow is choosing which matrices to expand and which block matrices to expand. This is mainly due to the size of the search space required. In this paper, we assessed the matrices ourselves by looking through the results of the algorithms. We believe a good method of implementing an algorithm that assesses these considerations and circumvents the issue of huge search space is to implement a random sampling-based search algorithm. We believe the method of Monte Carlo tree search would be useful [15]. We would need to implement a heuristic function to assess the quality of a high-dimension matrix. We pick one of the matrices $M \in \mathcal{M}_{12}$ where the permutation constructed from M has the following properties:

1. Weight of the path with the lowest differential and linear weights are close to each other and are high
2. Good differential/linear weight distributions
3. Good algebraic degree analysis results

we can then apply the Monte Carlo tree search to this matrix. The heuristic value should be also based on the properties mentioned above but with the additional consideration of cost. I.e., the number of 1s in the matrix. Since the heuristic value would be computationally inefficient for higher degrees, we think an algorithm that approximates the heuristic (such as a regression model based on the structure of the matrix) would work best.

We also believe that, when doing algebraic degree analysis on a permutation constructed from a matrix, in addition to measuring the degree itself, counting the number of

monomials in the bit with the highest degree could provide useful insight when choosing what matrix to consider for constructing a permutation.

A permutation made from these matrices needs to be assessed with the avalanche effect analysis [18].

We think further analysis of matrices that cause the same behavior as in Figure 8.7 and analyzing the implication proposed in Implication (9.1) could be useful insights.

Acknowledgements

I would like to give my deepest thanks to Professor Joan Daemen and PhD-candidate Jan Schoone for their unwavering support. Their continuous support and caring attitudes have made the process of writing my thesis truly comforting. They helped me understand concepts I had not heard before and they were patient enough to explain things multiple times if I did not understand initially. They were always open to my questions and they provided concise feedback throughout the writing process of my thesis. I appreciate their understanding It has been a very pleasant year of research and documenting alongside you. I am glad to have been supervised by you for the first long-form research document of my life. See you in the following years.

Bibliography

[1] John A Beachy. Abstract algebra (3_e) john a beachy and william d blair waveland press, inc.(2006) isbn 1-57766-443-4. pdf. page 120, 2006.

[2] Eli Biham and Adi Shamir. Differential cryptanalysis of des-like cryptosystems. *J. Cryptol.*, 4(1):3–72, 1991.

[3] Claude Carlet, Yves Crama, and Peter L. Hammer. Vectorial boolean functions for cryptography. In Yves Crama and Peter L. Hammer, editors, *Boolean Models and Methods in Mathematics, Computer Science, and Engineering*, pages 398–470. Cambridge University Press, 2010.

[4] Joan Daemen. Limitations of the even-mansour construction. In Hideki Imai, Ronald L. Rivest, and Tsutomu Matsumoto, editors, *Advances in Cryptology - ASIACRYPT '91, International Conference on the Theory and Applications of Cryptology, Fujiyoshida, Japan, November 11-14, 1991, Proceedings*, volume 739 of *Lecture Notes in Computer Science*, pages 495–498. Springer, 1991.

[5] Joan Daemen. Two ways of building round functions for block ciphers. Šibenik summer school lecture, 2016.

[6] Joan Daemen, Bart Mennink, and Jan Schoone. Introduction to Cryptography Lecture Notes 2019-2023. January 2023.

[7] Howard Whitley Eves. *Elementary matrix theory*. Courier Corporation, 1980.

[8] Tomás Fabsic, Otokar Grosek, Karol Nemoga, and Pavol Zajac. On generating invertible circulant binary matrices with a prescribed number of ones. *Cryptogr. Commun.*, 10(1):159–175, 2018.

[9] Dan Kalman and James E. White. Polynomial equations and circulant matrices. *Am. Math. Mon.*, 108(9):821–840, 2001.

[10] Serge Lang and Serge Lang. Groups. *Algebra*, pages 8, 546, 2002.

[11] Mitsuru Matsui and Atsuhiro Yamagishi. A new method for known plaintext attack of FEAL cipher. In Rainer A. Rueppel, editor, *Advances in Cryptology - EUROCRYPT '92, Workshop on the Theory and Application of of Cryptographic*

Techniques, Balatonfüred, Hungary, May 24-28, 1992, Proceedings, volume 658 of *Lecture Notes in Computer Science*, pages 81–91. Springer, 1992.

[12] Michael A Nielsen and Isaac L Chuang. *Quantum computation and quantum information*. Cambridge university press, 2010.

[13] James R Norris. *Markov chains*. Number 2. Cambridge university press, 1998.

[14] Thierry Simon, Lejla Batina, Joan Daemen, Vincent Grosso, Pedro Maat Costa Massolino, Kostas Papagiannopoulos, Francesco Regazzoni, and Niels Samwel. Friet: An authenticated encryption scheme with built-in fault detection. Cryptology ePrint Archive, Paper 2020/425, 2020. `https://eprint.iacr.org/2020/425`.

[15] Maciej Swiechowski, Konrad Godlewski, Bartosz Sawicki, and Jacek Mandziuk. Monte carlo tree search: a review of recent modifications and applications. *Artif. Intell. Rev.*, 56(3):2497–2562, 2023.

[16] Andrew J. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Trans. Inf. Theory*, 13(2):260–269, 1967.

[17] Senpeng Wang, Dengguo Feng, Bin Hu, Jie Guan, and Tairong Shi. Practical attacks on full-round FRIET. *IACR Trans. Symmetric Cryptol.*, 2022(4):105–119, 2022.

[18] Arthur F Webster and Stafford E Tavares. On the design of s-boxes. In *Conference on the theory and application of cryptographic techniques*, pages 523–534. Springer, 1985.

[19] Daniel Zwillinger and Stephen Kokoska. *CRC standard probability and statistics tables and formulae*. Crc Press, 1999.

The graph visualization website we used: `https://graphonline.ru/en/` Last accessed 2024-08-18