CHALMERS
UNIVERSITY OF TECHNOLOGY

Framework for Hidden Sum Attacks on Blockciphers

Constructing and Launching Hidden Sum Attacks Against Blockciphers

Master's thesis in Algorithms, Language and Logic

LUDVIG BLOMKVIST

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2021-22
www.chalmers.se

Master's thesis 2021-22

Framework for Hidden Sum Attacks on Blockciphers

Constructing and Launching Hidden Sum Attacks Against Blockciphers

LUDVIG BLOMKVIST

Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden 2021-22

Framework for Hidden Sum Attacks on Blockciphers
Constructing and Launching Hidden Sum Attacks Against Blockciphers
LUDVIG BLOMKVIST

Supervisor: Carlo Brunetta, Simula UiB, Bergen (Norway)
Examiner: Andrei Sabelfeld, Department of Computer Science and Engineering

Master's Thesis 2021-22
Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2021-22

Framework for Hidden Sum Attacks on Blockciphers
Constructing and Launching Hidden Sum Attacks Against Blockciphers
LUDVIG BLOMKVIST
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

A blockcipher is a deterministic algorithm that is operates on a fixed number of bits, called blocks. They are used as a fundamental crypto-primitive in security protocols and secure applications, from hashing algorithms to cryptographic protocols.

Cryptanalysis is the formal study of security and attack procedures on ciphers of all kinds. The fundamentals of the cipher used in this thesis, namely blockciphers were presented in 1949 by Claude Shannon and are still used today. By using methods of cryptanalysis it may be possible to detect both intended and non-intended trapdoors in blockciphers. Trapdoors are some kind of concealed property of the cipher that poses a threat to its integrity. One particular trapdoor is called a hidden sum. A hidden sum is a concealed property of the cipher that allows an attacker to compute the message from a ciphertext in a way that was not intended. A specific hidden sum that we will work with comes from Marco Calderini's PhD thesis, which asks the question *"Security for blockciphers is based on the idea that there is non-linearity. But linearity depends on the operator used. What if we use a different operator with a similar property to that of the original one? Will now the blockcipher be linear?"*. This led to more work, in particular Carlo Brunetta's master's thesis for finding these vulnerable blocks. However, there is no framework to create, store and use the operations needed to create the related attack.

This thesis has the goal to collect all the hidden sum theory into a python framework that can be used in both academic and research surrounding the hidden sum vulnerability. To conclude, the thesis will discuss open problems, missing development goals, and limitations. The framework is publicly available at the GitHub repo found at `https://github.com/ludblom/CircMod`.

Acknowledgements

Thank you Carlo for guiding me during the development of this thesis. I also appreciate all the help you have given me to understand mathematical principles I would have had a hard time understanding on my own.

Ludvig Blomkvist, Gothenburg, Jan 2022

Contents

List of Figures

1
Introduction

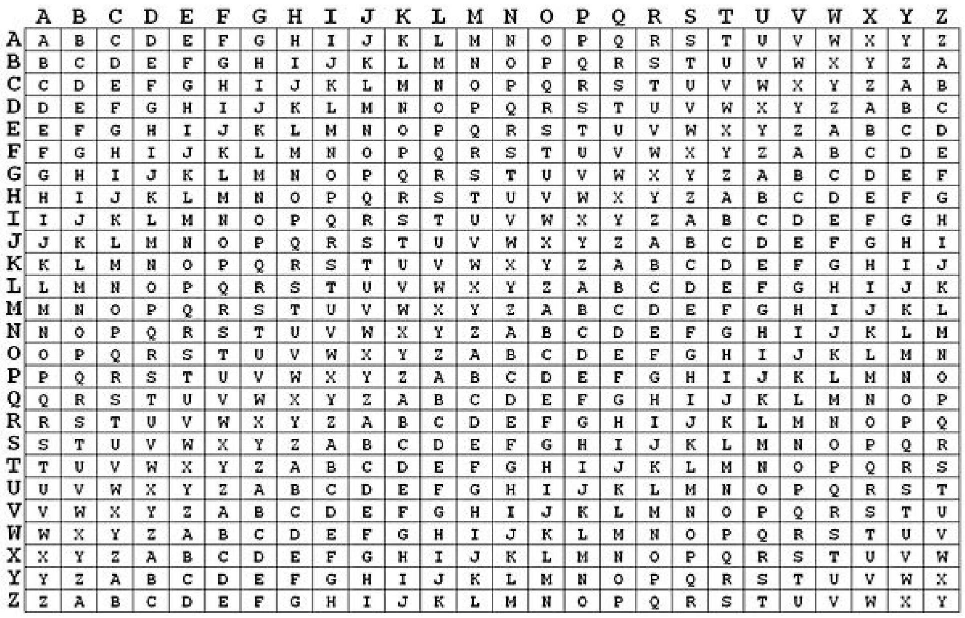
Our society revolves around cryptography, whether we know about it or not. When we log into our bank, social media account, and many other, cryptography is what keeps us safe. Because of this, we can conclude our society today evolves around encryption, we also have to verify and argue that a cipher still is holding up to this day and are not insecure.

As far back as the Roman empire, Julius Caesar used a particular encryption scheme to move commands across the battlefield [1]. Today we call this encryption scheme the *Caesar Cipher* and it works by moving each letter a specific number of steps, also called a monoalphabetic cipher [1]. Julius Caesar for example used three steps as his key, but any number of steps can be used. During this time and place, the encryption scheme worked well but we all understand that a cipher like that would not be secure today. One technique would be to apply frequency analysis and then inspect which characters or words we would expect to be most frequent based on the language used (for example the letter "e" or the word "the" in the English language) or simply brute force every possible permutation which at worse would be every letter in the alphabet.

In the 16th century the *Vigenere Cipher* instead was discovered by Blaise de Vigenere, also called a polyalphabetic cipher [5]. The cipher was based on the Caesar Cipher but with one major difference, it used a key. The Vigenere cipher was a big improvement compared to the Caesar cipher because by switching the key, each letter would be different in each place. How it works is by taking a string and a key then using the tableau you can see in Figure 1.1 to exchange the letter in the plaintext in the y axis and the key in the x axis, the cipher character is then the matching letter in the table.

The Vigenere Cipher provided better security compared to the Caesar Cipher but was still broken by Charles Babbage and Friedrich Kasiski [2] [5]. The problem with this attack was however the fact that the key repeats itself, so if a message is encrypted that is longer than the key, the key starts over. If an attacker figures out the length of this key it is much easier to determine what the key is, using for example frequency analysis like the Caesar Cipher. However, if the key is longer than the message then this encryption scheme is unbreakable since it upholds perfect secrecy [8].

Between the first and second World War is primarily when the world switched from classical cryptographic schemes (like the Vigenere Cipher) to more modern cryptographic methods [2]. During World War I people were still needed to encrypt and decrypt messages in between sending the message over the telegraph [2]. They also still used a version of the Vigenere Cipher, called the Vigenere Disk, along with



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

Figure 1.1: Vigenere Cipher Tableau.

codebooks and various manual methods [6]. During World War II, technology was introduced that automated this process more and more. The Germans for example had the notorious *Enigma Machine*, which was at the time a wounder that was thought to be unbreakable in the cryptographic field [2]. The military version of the enigma machine was broken not because of a flaw of the encryption, but because of how it was used. Some messages were predictable and capturing these predictable messages and using router computers that tested the combinations of the router wheels that fit that of the ciphertext was possible [3]. This in turn made it possible to break the key for each day.

After world war II, computers were used more and more in warfare and society. In the 1970's IBM needed security for their products so they developed an encryption scheme called Lucifer that later became DES [4]. This was one of the first modern, widely used ciphers for computers. At the turn of the century, the NSA however had decided that DES no longer was secure and suggested switching to AES, which we still use to this day [4]. Erlier than this, as far back then 1949, Claude E. Shannon proposed the idea that it is possible to build a robust encryption scheme from building blocks that by themself do not provide strong security through diffusion and confusion in a classified paper [8]. This theory is what paved the way for modern blockciphers. Blockciphers are deterministic algorithms that take in symmetric keys and a fixed number of bits from the data to encrypt or decrypt, called blocks. Each round also embed a secret into the cipher that prevent a user without the key from deriving the message from a ciphertext.

Apart from encryption, the idea that composes the blockcipher is also used for hash functions. Hashing is used heavily for protecting passwords in a database, keeping track of transactions in a blockchain like bitcoin, and much more. So it is an essential part of modern cryptography. Because of its relevance in the contemporary world, all possible attacks on the cipher have to be studied very closely to prevent a cipher with an unknown backdoor to be used.

This master thesis will discuss and present a particular vulnerability called hidden sum attacks. What we mean by a hidden sum attack is that there exists a secret vulnerability in the cipher that will allow us to compute the message from the ciphertext in a way that was not intended. To achieve this the content have been divided up into a few main chapters:

1. **"Preliminaries"** presents all the basic algebra needed, general information about the construction of blockciphers, and notations about modern techniques for cryptanalysis. The special rules for boolean algebra that are needed to know to work with the mathematics used in cryptography are also introduced.
2. **"Hidden Sum"** describes the construction of the hidden sum attack. The construction of the $\circ$-product is the first part that is presented followed by its special properties and rules. Following the creation of the $\circ$-product is how to use it to construct all the other parts needed to perform the hidden sum attack.
3. **"Framework"** goes through the development process of the framework, how it works and all the relevant functions it consists of. Some development choices, like the language and calculation of the dot product, are also discussed.
4. **"Discussion"** discusses some of the final results and thoughts on the result of the framework.

The framework is publicly available at the GitHub repository found at `https://github.com/ludblom/CircMod` under a GPLv3 license to ensure that the codebase stay open and free to use for everyone [9].

1.1 Related Work

The main goal of the thesis is to develop a framework that can be used to construct blockciphers vulnerable to the hidden sum attack and be able to launch attacks. The primary work that we will be working with is Marco Calderini and Carlo Brunetta's work. Calderini proposed this vulnerability and found out that it was possible [10]. Carlo worked to find algorithms and more efficient methods in finding these vulnerabilities [7]. This will be the primary work that the framework will be built around. With the framework in place, it closes the circle by providing an implementation of the theory.

2

Preliminaries

In this chapter, the basics of boolean algebra will be presented along with the construction of blockciphers and modern cryptanalysis.

2.1 Boolean Algebra

Boolean algebra is the branch of mathematics that only deals with truth values, *true* (1) and *false* (0). The main difference between boolean algebra and elementary algebra is the set of elements used. Since boolean algebra only deals with 0 and 1 it also uses different operations compared to elementary algebra, namely OR, AND, and XOR. For example, when computers perform $1 + 1 = 2$, they perform XOR and carry the bit over from position 0 to position 1. In boolean algebra, however, the overflowing bit is ignored, so $1 \oplus 1$ instead becomes 0.

p	q	$p \vee q$	$p \wedge q$	$p \oplus q$
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

Figure 2.1: Truth table for OR, AND and XOR.

Similar to the operational order of multiplication and addition in the natural numbers, we also assign a precedence order for AND, XOR, and OR. This is important to know which evaluation to do first without having to use parentheses. For example, $1 \vee 0 \oplus 1$ is either 0 or 1 depending on which operator is used first. But with a defined precedence table, it is easy to know which operation to start with.

Level	Operator
3	$\wedge$
2	$\oplus$
1	$\vee$

Figure 2.2: Precedence table for binary operators.

To convert a binary number to a decimal value, each digit is assigned a 2^n based decimal value. The decimal conversion is obtained by summing up all the powers of 2 values from position 0 to n. For example, a binary number b with the length of n

$$N = \sum_{k=0}^{n} b_k 2^k$$

Equation 2.1: Converting a binary number b of length n from binary to decimal.

$$b(N) = \begin{cases} 0, & \text{if } N\%2 = 0\text{, append to list and } b(N/2) \\ 1, & \text{if } N\%2 = 1\text{, append to list and } b(N/2) \\ & \text{otherwise if } N = 0\text{, stop} \end{cases} \Rightarrow \begin{pmatrix} x_0 & x_1 & \dots & x_n \end{pmatrix}$$

Equation 2.2: Converting a decimal number N to a binary vector b.

is converted to a decimal number N using Equation 2.1. A decimal number N can instead be converted to a binary number b using Equation 2.2.

To correctly convert a binary value to a decimal number it is also important to know which representation is used. In this paper, all calculations and binary numbers will be presented in little-endian order. Since the binary number in our case does not have to be split up and stored into bytes in memory, all that matters is if the most significant bit is on the left or right side. In the little-endian that we use, the most significant bit is on the right.

$$\begin{tikzcd} \lbrack 1 \arrow[rrr, bend right] & 1 & 0 & 1 \rbrack \end{tikzcd} \Rightarrow 1 + 2 + 8 = 11 \tag{2.1}$$

$$\begin{tikzcd} \lbrack 1 & 1 & 0 & 1 \rbrack \arrow[lll, bend left] \end{tikzcd} \Rightarrow 1 + 4 + 8 = 13 \tag{2.2}$$

Figure 2.3: (2.1) Little-endian, the most significant bit is to the right and the least significant bit to the left evaluates to 11. (2.2) Big-endian, the most significant bit to the left and least significant bit to the right, evaluates to 13.

Performing binary matrix multiplication and addition are also essential. Binary multiplication and addition are the same as regular multiplication and addition, except that the calculations differ slightly. When performing addition in base 10 for example, $1 + 1 = 2$. In binary XOR can be used as the addition operator instead. Binary-multiplication is the same but uses the $\wedge$ operator instead of multiplication. Matrix multiplication connects these two ways of performing addition and multiplication; take the example

$$\begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 \wedge 1 \oplus 0 \wedge 0 \oplus 1 \wedge 1 \\ 1 \wedge 1 \oplus 1 \wedge 0 \oplus 1 \wedge 1 \\ 0 \wedge 1 \oplus 1 \wedge 0 \oplus 0 \wedge 1 \end{bmatrix} \Rightarrow \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

It is done the same way as normal matrix multiplication but, as already stated, instead uses binary operators.

2.2 Vector Space

Calculations used are primarily represented as vectors. Vectors can also just be presented as numbers. For example to represent 0 to 3 the numbers can be represented as both a number and a vector.

$$0 = \begin{bmatrix} 0 & 0 \end{bmatrix}$$
$$1 = \begin{bmatrix} 1 & 0 \end{bmatrix}$$
$$2 = \begin{bmatrix} 0 & 1 \end{bmatrix}$$
$$3 = \begin{bmatrix} 1 & 1 \end{bmatrix}$$

Apart from one dimensional vectors we can also construct two-dimensional vectors using the same logic. An example denoted as A shows how it would look like.

$$A = \begin{bmatrix} 7 \\ 5 \\ 2 \end{bmatrix}$$

We from A at least know that it has 3 rows, but the number of columns is not defined. Because of this, we will always define the size in all directions so that A can be converted into a binary vector. In the case we have written that $N = 3$, we know that the matrix will be a 3×3 matrix and can with this information convert the vector to binary.

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

When preforming operations in ciphers or calculating vectors will most often be used. Knowing that the vectors represent values in base 10 is essential in order to be able to use and work with the theory.

2.3 Calculate the Inverse Matrix and Verifying Linearity

To find the inverse matrix we will use modified versions of the Gauss-Jordan elimination scheme [17]. Throughout this thesis, we have special notations to denote linearity. When we talk about something being linear it will be denoted as GL_+ for linearity in XOR and $GL_\circ$ for being linear in the $\circ$ operator. The inverse matrix is used for the λ matrix to perform the decryption but also used to check if a specific matrix is linear or not, i.e. verify that $\lambda \in GL_+ \cap GL_\circ$. To calculate the inverse using the Gauss-Jordan elimination we take a matrix A and append the identity matrix I of the same dimension. The goal is to convert A into an identity matrix I using row multiplication and row shifts while doing the same operations on the I matrix. If A can successfully be converted to I the original I will be the inverse of A, written as A^{-1}.

$$[A|I] \Rightarrow \left[I|A^{-1}\right]$$

The operation used to calculate the inverse decides which dimension it is linear in. For example, by default XOR is used since we are in base 2.

$$\left[\begin{array}{cc|cc} 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{array}\right] \xrightarrow{R_2 \oplus R_1} \left[\begin{array}{cc|cc} 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{array}\right]$$

This will give us both the inverse and also verify that the matrix A is in GL_+. It is possible to modify and use some other operator besides XOR to verify if it is linear in that domain, this is exactly what we do to verify if the matrix also is in $GL_\circ$, we switch from using XOR to $\circ$.

Finding the inverse using XOR resp. $\circ$ is reported in Appendix A.1 resp. A.2. The implementation using XOR is executed with time complexity of $O(n^3)$ while using the ring is executed in $O(n^5)$. The increment of time complexity is primarily because we have constructed the ring operator for every combination of two binary numbers a and b, hence the complexity increases when calculating it compared to using XOR.

2.4 Security Breach Severity

When a security vulnerability is found in a system there are several levels of how severe the vulnerability is. In network security, for example, the biggest breach of security is for an unauthorized user to gain root access to the servers. Underneath several other layers of less severe security breaches exist. It is therefore important to determine how severe a particular security breach is.

Let's discuss the different levels of security breaches for blockciphers [16]. The primary goal when attacking a blockcipher is to get the secret key. This is because the attacker then can decrypt all the messages that have used the key and also create newly fabricated ciphertexts. The next step is a Global Deduction vulnerability. This would be to find an algorithm A that is equal to performing $E_K(..)$ or $D_K(..)$ without having the key K. Information Deduction is a process where some type of information about the key or message is known that was not known before the attack. The least severe case is that the attacker can tell the algorithm used when encrypting. So each step decreases the severity but everyone gives the attacker some kind of information about the ciphertext.

What we can observe is that a security breach does not always mean finding the secret key K. That is the most severe security breach of them all. Information about a cipher can potentially be figured out without knowing or deriving the key that still can compromise the integrity of the cipher.

The attack discussed in this thesis will eventually be able to calculate a sum to use to decrypt and encrypt messages. So the attack itself is not a total break, which would mean to be able to calculate the key, but instead allow us to use the sum to do it.

2.5 Blockcipher and Cryptography

A blockcipher is a cryptographic scheme that is built on the linear properties of all blocks. Linearity indicates that there exists an inverse that makes it possible to move in both directions using the inverses of every part.

We will also use specific notations for the different blocks

- λ: The mixing layer (P-box)
- γ: The substitution layer (S-box)
- T_+: The key layer
- Δ: The entire blockcipher

In the thesis, the entire cipher connected with the substitution layer, mixing layer, and the key layer will be called Δ. If particular instances are discussed around the cipher an algebraic abstraction may be used by the notation $\Delta =< \gamma\lambda, T_+ >$. This can be seen as Δ but with the mixing and permutation parts, $\gamma\lambda$, and the key layer, T_+, as individual entities.

The type of blockciphers studied in this thesis is iterated ciphers that are computed using a set number of rounds, N. During each of the rounds, the cipher performs three operations, the first of which is performing a non-linear substitution on the plain text. The operation we use for this is the substitution layer, γ. What this operation provides for the cipher is the "confusion" part according to Shannon's theory of *Confusion and Diffusion* [8].

AES, one of the most famous blockciphers and still used today, is designed to perform a different amount of rounds based on the size of each block. The number of rounds is 10 rounds for 128-bit keys, 12-rounds for 192-bit keys, and 14 rounds for 128-bit keys [18]. Generally, the security of the cipher increases with the number of rounds, but it also becomes slower and costs more computing power. So when designing a blockcipher, comparing the number of rounds with the increase in security is an important part to design a secure and efficient blockcipher.

2.5.1 Building Blocks of the Cipher Used

A blockcipher consists of pre-specified mixing and substitution boxes that modify the cipher and key a pre-defined number of times, called rounds.

The substitution box (S-Box or γ) exchanges a specific number to another number based on a pre-defined table, covering Shannon's confusion principle of creating blockciphers [8]. It is constructed by creating a dictionary of all combinations of the total number of bits the block of the cipher consists of and mapping each number to another number, with the constraint of it being a one-to-one bijection. This property ensures that a value x always substitutes to y, hence also y substitutes to x. If x and y instead substitutes to p, when preforming the decryption it is impossible to know if p goes to x or y.

The mixing layer (P-Box or λ) covers Shannon's principle of confusion when creating a blockcipher [8]. The matrix is created by generating a list with 0 or 1 in

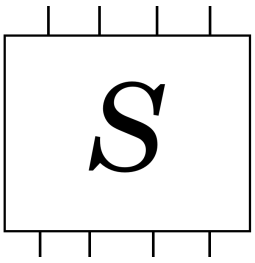


Figure 2.4: The S-Box.

each position, creating a binary matrix that is the same size as the block. An inverse matrix is then calculated to be used for decryption. The framework calculates this inverse matrix using a modified version of Gaussian elimination to work with binary numbers [12].

If the user allows the framework to create the matrix, then it will simply randomly generate matrices and calculate its inverse, if it finds the inverse it is done. When the user submits a mixing layer the framework calculates its inverse and throws an exception if the matrix submitted is not linear (i.e. invertible).

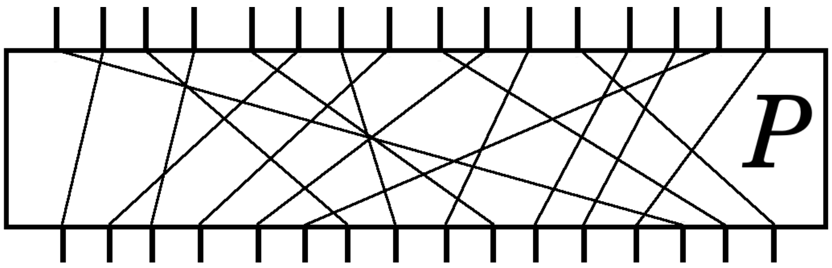


Figure 2.5: The P-box.

The key box is used similarly to the S-Box, with the same rules but instead used to generate different key rounds. Since the key size is the same as the block size, the key box generates all the combinations of as many bits. The data and key are then XOR'ed with the data before passing through the S and P-boxes. Then a new round is generated for the next iteration.

2.5.2 Encryption and Decryption Phase

The encryption and decryption of data pass through the S and P-box while XOR is executed on the data along the way using the different key rounds. Depending on how many rounds the cipher has been initialized with will decide how many times the data goes through this process, see Figure 2.6. The decryption function works the same way but generates the last key from the encryption phase and then does everything in reverse.

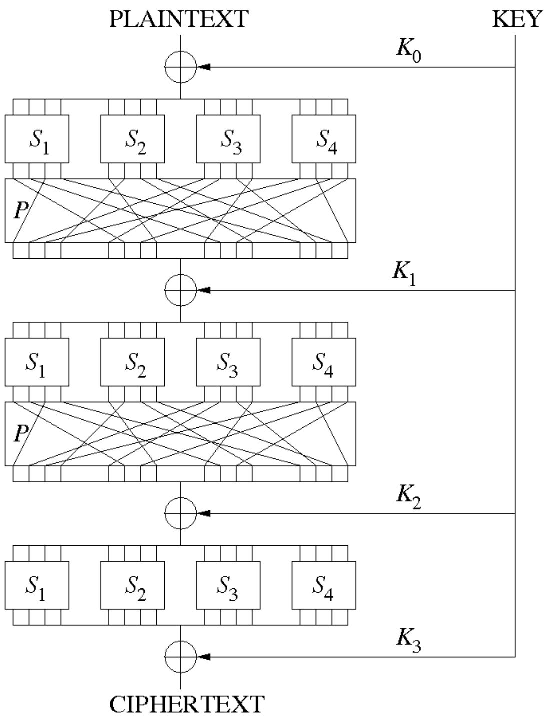


Figure 2.6: A 2-round blockcipher complete with mixing layer, substitution layer, and key layer.

2.6 Cryptanalysis

When a blockcipher is implemented they are most often also subject to rigorous cryptanalysis. Cryptanalysis is a different way of analyzing a specific cipher and trying to gain knowledge about how the cipher work. There are numerous different strategies to perform cryptanalysis, which strategy to use depends on the cipher that is being analyzed. In the coming section, a few cryptanalysis techniques will briefly be introduced and discussed to give the reader an introduction to how intentional or non-intentional vulnerabilities in ciphers are discovered.

Linear Cryptanalysis

Linear cryptanalysis is a method where an attacker finds affine approximations to the parts constructing the cipher (i.e. λ and γ). The technique is attributed to Mitsuru Matsui who used it in 1992 to attack the FEAL-6 cipher [24]. Because of the property of XOR, any linear equation for cipher and key bits should hold with a property of 50%. However, using linear cryptanalysis it is possible to find areas where this does not hold. Meaning that the probability a particular bit becomes 0 or 1 on a particular position is more or less than 50%.

Differential Crypanalysis

Eli Biham and Adi Shamir are generally attributed for the discovery of the differential cryptanalysis attack, even though it already was known by parties like the NSA and IBM before they published, they, however, called it the "T-Attack" [23]. With differential cryptanalysis, the difference of input compared to output is studied to discover areas in the cipher where non-random behavior is present [19] [20].

Boomerang

The boomerang attack is based on the differential attack, the primary difference is that instead of one long differential, we have two shorter differentials, each half of the cipher [21].

Impossible Differential Cryptanalysis

Impossible differential cryptanalysis is another version of differential cryptanalysis. Impossible differential cryptanalysis differ from ordinary differential attack by instead of tracking differences that occur more often than would be expected, it tracks impossible differences (i.e. a probability of 0) [22].

New methods with varying success rates are discovered ever so often, and the methods that exist are frequently improved. Hence if a new cipher is developed it should be assumed that it will be meticulously analyzed and tested against the currently known attacks.

3 Hidden Sum

This chapter explains how to construct the hidden sum used to attack blockciphers. For the attack to work, the building blocks of the cipher must follow a specific set of rules. In particular, $\lambda \in GL_\circ \cap GL_+$ and $\gamma \in GL_\circ$, if this is the case, we can be sure that the attack works [7] [8]. To know if this is the case, the $\circ$ operator has to be defined. The $\circ$ operator is very similar to XOR but with a few values modified in a particular way. We provide examples for two different dimensions and present the construction of the $\circ$-sum while highlighting how it is generated. To understand how the operator $\circ$ works, a 5×5 matrix will be used to more easily visualize how k modify the matrix M and a 3×3 to outline the parts necessary to understand the concept.

3.1 Creating the Ring Product

The $\circ$-product is the operator that creates the hidden sum attack. This operator emulates the $\oplus$ operator on everything except for specific areas. This is done by crafting matrices that in a particular area differ from XOR when performing matrix multiplication. Describing this we can think of a vector a and a matrix M with a dimension of N. If $M = I$, meaning it is an identity square with the same size as a, then $a.M_0 = a$. This can be seen in Equation 3.1 which use the M matrix unmodified to create the XOR operator.

The $\circ$ operator however modify the upper right square of the M matrix. The matrix is modified based on a variable k that determine the size of the area modified and also the second value in the operation. The area of M which is modified for a selection of k can be observed in Equation 3.2. An important bound for the $\circ$-sum to work is $1 \leq k < N - 1$ that limits how big or small N can be. Because for $1 \leq k < N - 1$ to be possible, N must be greater or equal to 3.

$$a + b = aM_0 + b = a \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} + b$$

Equation 3.1: XOR defined the same way as the $\circ$-operator.

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

Equation 3.2: The area modified when $N = 5$ for $k = 1..3$.

The way in which to construct the $\circ$-operator we have to start with the k variable and construct the matrices needed from it. If we have that $N = 5$ and $k = 2$ we start by checking so that it is possible, and since $1 \leq 2 < 4$ we can conclude that k is within the legal limit.

The $\circ$-operator looks like this, $a \circ b = aM_b + b$, so depending on what b is, it will modify what aM_b becomes before XOR on both values is applied [8]. As we mentioned before, the only part of M that is modified is the top right corner, with the size of k, following the structure of Equation 3.2. This matrix we will call B_v, where v represent the number of b in the example above. B_v in its essence is a matrix with the horizontal size of k combined from multiple other matrices from a base which is called $B_\circ$, which we will start with to construct.

Since we know that $k = 2$ for this example, we know that we will have 2 horizontal bits, hence all the possible decimal values a vector of that size can take is between 0 and 3.

$$\begin{aligned} 0 &\longleftrightarrow (0,0) \\ 1 &\longleftrightarrow (1,0) \\ 2 &\longleftrightarrow (0,1) \\ 3 &\longleftrightarrow (1,1) \end{aligned}$$

For the sake of simplicity we will use decimal values but they should be seen as vectors. To get the number of rows that is needed we calculate $N - k = 5 - 2 = 3$ which mean that we need three columns. With this information we can construct $B_\circ$. With the information of the size of $B_\circ$, we just have to know how to assemble it. The first rule is that we need a zero base, so lets start to just add a row of zeros in the base of $B_\circ$

$$B_\circ = \begin{bmatrix} 0 & x_{0,1} & x_{0,2} \\ x_{1,0} & 0 & x_{1,2} \\ x_{2,0} & x_{2,1} & 0 \end{bmatrix}$$

The next step is move row and column wise down the matrix and exchange $x_{a,b}$ with the biggest value in the size of k. Since k is 3, the first row and column will be 3, the next row and column will be 2 and so on until the matrix is populated.

$$\begin{bmatrix} 0 & 3 & 3 \\ 3 & 0 & x_{1,2} \\ 3 & x_{2,1} & 0 \end{bmatrix} \Rightarrow \begin{bmatrix} 0 & 3 & 3 \\ 3 & 0 & 2 \\ 3 & 2 & 0 \end{bmatrix}$$

No matter how big $B_\circ$ is, it always is structured in this way. As mentioned before, $B_\circ$ represent in a way the value of B_v, in particular the bit values of B_v. Each column represent each individual bit, so the first column represent the first bit, second column the second bit and so on.

$$\begin{cases} B_1 = \begin{bmatrix} 0 \\ 3 \\ 3 \end{bmatrix} \Rightarrow \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \end{bmatrix} \\ B_2 = \begin{bmatrix} 3 \\ 0 \\ 2 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 1 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} \\ B_4 = \begin{bmatrix} 3 \\ 2 \\ 0 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 1 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \end{cases}$$

But since $N = 5$ and we can see that we only have B_v for v equal to the first, second and third bit we need to also have to know what B_v will be for the bits not included. For these values, B_v will simply be zero, represented as I for the identity matrix. Constructing the M matrices for the bits in $B_\circ$ can now be constructed [7].

$$M_v = \begin{bmatrix} I_{n\times n} & B_v \\ 0 & I_{k\times k} \end{bmatrix}$$

$$\begin{cases} M_1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad M_2 = \begin{bmatrix} 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \\ M_4 = \begin{bmatrix} 1 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \end{cases}$$

What we can note is the top right corner where the B_v matrices have been placed as explained earlier. From this the $\circ$-operator can be constructed. A different notation is used to calculate $\circ v$ for each bit, called τ. τ is constructed with all canonical bases e for each bit.

$$\tau_b(x) = x \begin{bmatrix} I_{n\times n} & B_b \\ 0 & I_{k\times k} \end{bmatrix} + b$$

$$\begin{cases} \tau_1(x) = xM_1 + \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \end{bmatrix} & \tau_2(x) = xM_2 + \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \end{bmatrix} \\ \tau_4(x) = xM_4 + \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \end{bmatrix} & \tau_8(x) = xI + \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \end{bmatrix} \\ \tau_{16}(x) = xI + \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \end{bmatrix} \end{cases}$$

We can see that where the bit is undefined we only have an identity matrix I in its place since B_v only is zeros. Using this operator we can see that all the bits are set and can also be combined to create all the possible values. If a number is not defined for τ, it just use the zero base. So if B_5 is to be constructed, all needed to do is $\begin{bmatrix}1 & 0 & 0 & 0 & 0\end{bmatrix} + \begin{bmatrix}0 & 0 & 1 & 0 & 0\end{bmatrix} = \begin{bmatrix}1 & 0 & 1 & 0 & 0\end{bmatrix}$. with this, B_5 is constructed by $B_1 + B_4 = B_5$. Using B_5 we can now construct M_5 the same way as explained.

$$B_1 + B_4 = \begin{bmatrix}0 & 0\\1 & 1\\1 & 1\end{bmatrix} + \begin{bmatrix}1 & 1\\0 & 1\\0 & 0\end{bmatrix} = \begin{bmatrix}1 & 1\\1 & 0\\1 & 1\end{bmatrix} = B_5$$

Lets explore an example using the operator, say that $a = 3 = \begin{bmatrix}1 & 1 & 0 & 0 & 0\end{bmatrix}$ and $b = 5 = \begin{bmatrix}1 & 0 & 1 & 0 & 0\end{bmatrix}$. We then want to compute $3 \circ 5 = 3M_5 \oplus 5$, B_5 is already calculated from before so can simply insert it at the top of a 5×5 identity matrix to create M_5.

$$M_5 = \begin{bmatrix}1 & 0 & 0 & 1 & 1\\0 & 1 & 0 & 1 & 0\\0 & 0 & 1 & 1 & 1\\0 & 0 & 0 & 1 & 0\\0 & 0 & 0 & 0 & 1\end{bmatrix}$$

And last of which the ring operator can be calculated

$$3 \circ 5 = 3M_5 \oplus 5 = \begin{bmatrix}1 & 1 & 0 & 0 & 0\end{bmatrix}\begin{bmatrix}1 & 0 & 0 & 1 & 1\\0 & 1 & 0 & 1 & 0\\0 & 0 & 1 & 1 & 1\\0 & 0 & 0 & 1 & 0\\0 & 0 & 0 & 0 & 1\end{bmatrix} \oplus \begin{bmatrix}1 & 0 & 1 & 0 & 0\end{bmatrix}$$

$$= \begin{bmatrix}0 & 1 & 1 & 0 & 1\end{bmatrix} = 22$$

Lets compare this with the XOR operator to see the difference between the $\circ$ and XOR operator.

$$3 \oplus 5 = 6$$

We can hence clearly see that $3 \oplus 5 \neq 3 \circ 5$, so we can expect that their values differ between each other, especially when bits in B_v differ from zero.

3.2 Rules of the $\circ$-product

With the $\circ$-operator defined it is important to know in what way it relates to XOR. Both XOR and the $\circ$-product are abelian groups and because of this share the same rules amongst them, hence the rules we have to prove are the Associative, Commutative, and an existence of a zero element property.

Theorem. *Prove the associative property for any three vectors a, b, c in the same dimension.*

Proof.

$$\begin{aligned}(a \circ b) \circ c &= (a \circ b)M_c + c \\ &= (aM_b + b)M_c + c \\ &= aM_bM_c + bM_c + c \\ &= aM_bM_c + b \circ c \\ &= aM_{b\circ c} + b \circ c \\ &= a \circ (b \circ c)\end{aligned}$$

□

Theorem. *Prove the commutative rule for any vector a, b in the same dimension.*

Proof.

$$\begin{aligned}a \circ b &= b \circ a \\ aM_b + b &= bM_a + a \\ aM_bM_a + bM_a &= bM_aM_a + aM_a \\ aM_{b\circ a} + bM_a &= b + aM_a \\ aM_{b\circ a} + bM_a + a &= b + aM_a + a \\ aM_{b\circ a} + bM_a + a &= b + 0 \\ aM_{b\circ a} + b \circ a &= b \\ a \circ (b \circ a) &= b \\ a \circ a \circ (b \circ a) &= a \circ b \\ b \circ a &= a \circ b\end{aligned}$$

□

Theorem. *Prove the existence of zero elements for vector a.*

Proof.

$$\begin{aligned}0 \circ a &= 0M_a + a \\ &= a \\ &= a + 0 \\ &= aM_0 + 0 \\ &= a \circ 0\end{aligned}$$

□

3.3 Moving Between XOR and $\circ$ Universes

Having both XOR and $\circ$ defined we can inspect the differences between the two operations. In Figure 3.1 we have an example where $N = 3$, the red highlighted values that differ in between XOR and $\circ$.

$\oplus$	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	0	3	2	5	4	7	6
2	2	3	0	1	6	7	4	5
3	3	2	1	0	7	6	5	4
4	4	5	6	7	0	1	2	3
5	5	4	7	6	1	0	3	2
6	6	7	4	5	2	3	0	1
7	7	6	5	4	3	2	1	0

$\circ$	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	1	0	7	6	5	4	3	2
2	2	7	0	5	6	3	4	1
3	3	6	5	0	7	2	1	4
4	4	5	6	7	0	1	2	3
5	5	4	3	2	1	0	7	6
6	6	3	4	1	2	7	0	5
7	7	2	1	4	3	6	5	0

Figure 3.1: XOR compared to the $\circ$ operator when $N = 3$ and $k = 1$.

Both of these tables are constant in 3×3, so what we can conclude is that there should be a way to walk in between the XOR and $\circ$ universe. In particular, this is in fact the reason why both γ and λ have to be linear in the $\circ$ operator. So that when the ciphertext is created it will be in the XOR universe which is not linear in γ, but when moved into the $\circ$-universe is linear in both γ and λ. For this we will create a new map called ϕ. ϕ will have the property of both moving from XOR to $\circ$ and $\circ$ to XOR [7] [8]. The map is defined as

$$\phi\left(\sum_i x_i e_i\right) = \bigcirc_i x_i e_i$$

To create $\phi(3)$ for example, the vector looks like $\begin{bmatrix}1 & 1 & 0\end{bmatrix}$ which will look like $\phi(3) = 1 \circ 2 = 7$. So if we are in XOR and it is 3, moving into the $\circ$ universe it will be 7. Preforming this on all the values in N will create a map over the values, moving in the other direction simply invert all the values so that the values in the $\circ$-universe moves into the XOR universe. To keep track of which universe a variable is in we will define the variable by tilde. Taking a for example, if it is in the $\circ$-universe, it will be labeled as $\tilde{a}$ and if in the XOR universe, simply as a.

This will allow us to preform operations on the cipher text that is necessary in order to preform the operations needed to create the final sum. In particular, because of the definition we know that $\phi(a \oplus b) \neq \phi(a) \oplus \phi(b)$. However, because of the definition of linearity in $\circ$, we will be able to assure that $\phi(\tilde{a} \circ \tilde{b}) = \phi(\tilde{a}) \circ \phi(\tilde{b})$ [7].

3.4 Calculate the Message from the Ciphertext

The detachment of a message is the last step to present the vulnerability. For this, the construction of a matrix R^{-1} using both the cipher and the ϕ map have to be done. R first of which is the identity matrix of size N, then encrypted using Δ, each row is then exchanged to the value of ϕ, lifting it into the $\circ$ universe. XOR is then executed on each row together with the zero bases that have gone through the same operations.

$$\begin{bmatrix}0 & 0 & 0\end{bmatrix} \xrightarrow{\Delta} \begin{bmatrix}0 & 1 & 0\end{bmatrix} \xrightarrow{\phi} \begin{bmatrix}0 & 1 & 1\end{bmatrix} = \begin{bmatrix}0 & 1 & 1\end{bmatrix} = \tilde{V}_0$$

$$R = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{matrix} \xrightarrow{\Delta} \\ \xrightarrow{\Delta} \\ \xrightarrow{\Delta} \end{matrix} \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 1 & 0 \end{bmatrix} \begin{matrix} \xrightarrow{\phi} \\ \xrightarrow{\phi} \\ \xrightarrow{\phi} \end{matrix} \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 0 \end{bmatrix} = \begin{bmatrix} \tilde{V_1} \\ \tilde{V_2} \\ \tilde{V_3} \end{bmatrix} \Rightarrow \begin{bmatrix} \tilde{V_1} \oplus \tilde{V_0} \\ \tilde{V_2} \oplus \tilde{V_0} \\ \tilde{V_3} \oplus \tilde{V_0} \end{bmatrix}$$

R^{-1} can then be found using Gauss-Jordan elimination as described in Section 2.3, this also confirms that the R matrix is linear. The last step is to calculate the message from the ciphertext. The first step is to move over the ciphertext from c to the ring universe [7].

$$c \xrightarrow{\phi} \tilde{c}$$

The next step is to get the message

$$\tilde{m} = (\tilde{c} \oplus \tilde{V_0})R^{-1}$$

$$\tilde{m} \xrightarrow{\phi^{-1}} m$$

4
Framework

The framework is divided into different classes. The HiddenSum class that is used to perform the hidden sum attack, ToyCipher class that creates a relatively weak cipher that can be used to test the attack on, Operations class that provide the special operations needed to perform the attack, and the Matrix class that provides matrix operations. This chapter is going to present how the different parts are built, what they consist of, how they work and provide a few examples.

The goal of the framework is to create all the tools needed to work with boolean algebra, create toy ciphers and be able to implement and use a hidden sum vulnerability into the cipher. When designing the framework a few efficiency compromises had to be made. The primary compromise was to create a matrix and manually perform matrix multiplication. A more efficient way would be to take a list with the input and then directly perform a modulo calculation over the matrices. For example, take the vectors A and x. Below we present two equivalent ways of performing matrix multiplication.

$$x.A = \begin{bmatrix} x_0 & x_1 & x_2 \end{bmatrix} \begin{bmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$x.A = \begin{cases} x_0 A[0][0] \oplus x_1 A[1][0] \\ x_1 A[1][1] \\ x_0 A[0][2] \oplus x_2 A[2][2] \end{cases}$$

4.1 Overview of the Framework

The reason for going with the first approach is that we have to do more than simply perform matrix multiplication. A user should be able to provide different block sizes, only provide the λ and the framework compute the inverse and a way to save the matrix for later use. Because of this, speed is not as important as usability. When a matrix has been found and the user wants to implement the vector A into a real blockcipher, the first matrix can easily be converted into a second one instead.

The goal of the framework is to be able to provide the basic functions needed to work with binary matrices, create the $\circ$-product, create and use toy ciphers and be able to launch attacks against them.

Without the added complexity of T_+ however, creating and understanding the attack becomes much easier. To start with the actual creation of vulnerable λ and

γ pairs will have to be created by hand since the ToyCipher class only generate invertible matrices that not necessarily are vulnerable.

As already stated, the framework is divided into four sub-classes; HiddenSum, Matrix, Operations and ToyCipher. Each of these classes handles specific areas of the framework and has been divided up to simplify accessing only the parts needed for a specific task.

The ToyCipher is built by inheriting 4 different classes, KeyBox, SBox, PBox and Matrix.

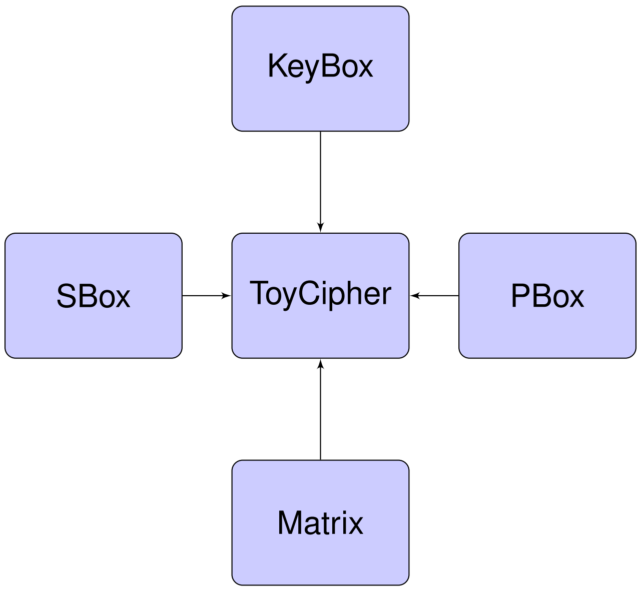


Figure 4.1: The ToyCipher connected with its sub-classes SBox, KeyBox, PBox, and Matrix.

The HiddenSum class only inherent Matrix but also use Operations and the ToyCipher class. Worth noting is that the ToyCipher class is completely detached from the HiddenSum class.

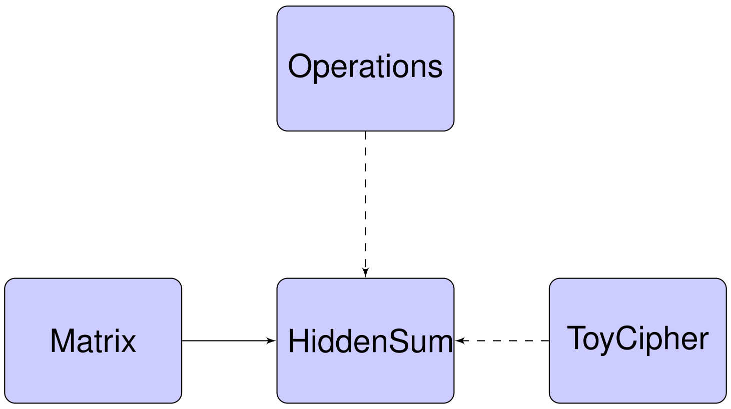


Figure 4.2: The HiddenSum class is connected to Matrix and uses the classes Operations and ToyCipher.

4.1.1 The Python Programming Language

Python is the language of choice for this framework because it is freely available, easy to understand, and easy to use. It is an interpreted high-level general-purpose programming language created in 1991 by Guido van Rossum. Python is however notorious for being quite slow. Creating the framework in C or C++ would be optimal, but that would also force users of the framework to use that language, which would reduce its usability.

However, worth noting is that this framework is written from scratch. It does not have any dependencies outside of the default packages Python includes. Hence a subset of the Python language that aims more towards execution speed could, and probably should, have been used. A language such as this is Cython. Cython aims to be a subset of the Python language with a similar syntax but aims to have an execution speed similar to the C language. It is also then possible to import the code in regular Python source code, thus allowing a user of the framework to still use a high-level and easy-to-use language like Python but also benefit from the speed of C. The fact that no other packages that we would have liked to use would conflict with Cython was unknown in the beginning, so Python was the right way to go to avoid a potential rewrite of most if not all parts of the framework.

4.2 HiddenSum

The HiddenSum class is used to construct an attack against a blockcipher Δ. Important to note is that the class only takes in ciphers from the class ToyCipher, so if a cipher can not be constructed in terms of the ToyCipher class, this class most probably can not be used. The HiddenSum class contains almost everything needed to launch the attack, such as ϕ, ϕ^{-1}, M and M^{-1}. The $\circ$ operator is however imported from the Operations class. The class is defined by the functions listed below along with a brief description about what they do.

`__is_attackable()`
Check that γ and λ are attackable.

`__check_S_attackability()`
Check the γ box attackability.

`__lambda_check()`
Make sure that lambda are in XOR and $\circ$.

`change_key(key)`
Update the class stored M matrix for a new key.

`lambda_tilde(P)`
Generate the matrix from λ.

`lambda_GL_ring(A_t)`
Calculate that $GL_\circ$ of A_t is true.

`create_M(t)`
Create the M matrix from the ToyCipher class.

`attack(c, key)`
Decrypt the cipher c using either the pre stored R for a specific key or generate a new R for a new key.

When a HiddenSum class is created it takes in N and k used to crate the $\circ$ operator from the Operations class. It also takes in the ToyCipher and a *key*. In case the ToyCipher is not given to the class the user is left responsible to give it at a later stage. No checks will necessarily then be made to make sure that the ToyCipher is attackable. The reason for this is if the user wants to test different kinds of cipher constellations that not necessarily is vulnerable, it is possible to do so.

4.3 ToyCipher

The framework includes a ToyCipher class that can be used to create blockciphers of different sizes and rounds. This was done to simplify executing a hidden sum attack and working with different versions. It is built with many cornerstones that modern blockciphers, such as rounds, mixing and substitution layers, and so on [11].

To use the toy cipher two parameters have to be given, the length of the block and how many rounds the cipher should have. The framework will then generate new P and S-boxes at random. Because it is at random, the cipher is not necessarily attackable. The user will create the vulnerable λ and γ boxes either by hand or using the HiddenSum class to construct them. The available function the ToyCipher class have is

`__binary(b)`
: Make sure a list or string is binary.

`__check_input(data, key)`
: Check that the inputs are correct.

`__load_p_box(orig_indent, content)`
: Load λ from a file.

`__load_s_box(orig_indent, content, box)`
: Load γ and Key box from file.

`__preform_splitted_substitution(data, encryption)`
: Preform data substitution using multiple γ.

`save_cipher(file_name)`
: Save the current cipher to a file with a specific name.

`load_cipher(file_name)`
: Load a cipher at a specific position in the file system.

`encrypt(data_t, key_t)`
: Encrypt the data with a key.

`decrypt(data_t, key_t)`
: Decrypt the data with a key.

The cipher itself is not built to pose a threat of any means, but this does not mean that it should be used in production. It is simply a prototype used in this thesis to help the user of the framework to understand how the hidden sum attack works, but many more vulnerabilities may be found! Real blockciphers are crafted with many precautions in mind where the λ and γ-boxes have been thoroughly monitored and tested to not pose the threats that are described in this thesis using various methods, such as linear and differential cryptanalysis as briefly described in Section 2.6 or further formally described in Knudsen-Robshawr's book [11].

4.4 Matrix

The Matrix class contains all the operations regarding matrices. This class is inherited by every other class since operations on matrices are such a central part of the framework, which is the reason for it to be developed in its own class. The functions the class contains is

`int_to_binary(i, l)`
: Convert an integer to binary list.

`binary_to_int(b)`
: Convert a binary list to an integer.

`matrix_sum(a, b)`
: Sum two matrices.

`get_identity(n)`
: Generate an identity matrix.

`row_shift(M, x, y)`
: Shift the row of a matrix.

`matrix_mul_row(M, x, y)`
: Multiply row x of a matrix M with a vector y.

`matrix_mul_row_column(x, M)`
: Preform matrix multiplication for a one dimensional vector with a multi-dimensional vector.

`calculate_inverse(A_t)`
: Calculate the inverse of a matrix A_t using Gauss-Jordan elimination.

`xor(a, b)`
: XOR two lists together.

4.5 Operations

The Operations class has all the special operations defined. The reason these are in a class of their own and not a part of the HiddenSum class is because the user may want to use the operations without creating a hidden sum vulnerability, the prime reason for this is to perform boolean algebra using the new operators.

`__generate_Bex()`
: Generate B_v matrices.

`__generate_Bo(N, k)`
: Generate a B_o matrix.

`phi_pos_a(a, P_a)`
: Determine $\lambda_\circ$ at position a using the $\circ$-product.

`phi_pos_a_xor(a, P_a)`
: Determine λ_+ at position a using xor.

`phi_map(P)`
: Calculate the ϕ and ϕ^{-1} map.

`get_Bx(b)`
: Get the M_i matrix.

`ring(a, b)`
Preform the $\circ$-operation.

`dot(a, b)`
Preform the dot product.

`matrix_mul_row_ring(M, x, y)`
Matrix multiplication using $\circ$.

4.6 Usage of the Framework

Simply using the ToyCipher class can be seen in Listing 1. This creates a random λ that is linear in XOR and a random γ. Because of this, the cipher is not necessarily vulnerable to the attack.

```
from ToyCipher.ToyCipher import ToyCipher

t = ToyCipher(block_len=3, rounds=5)
c = t.encrypt("010", "101")
m = t.decrypt(c, "101")

print(m)
```

Listing 1: Create a 3×3 ToyCipher and encrypt/decrypt a message then print it out.

Performing the HiddenSum attack is done similarly. In the first version of the cipher, a vulnerable λ and γ have to be provided. In Listing 2, the HiddenSum class is initialized in two ways. *hs1* is provided with the *key*, this will set up the R matrix that is used to perform the attack for that key, making it unnecessary to recreate it every time a new cipher using the same key need to be decrypted. In case the user wants to recreate R, create the HiddenSum like done in *hs2*.

This works well for when we have a vulnerable cipher. However, finding these ciphers is not a trivial task. One way may be to just generate a random λ and γ, see if $\lambda \in GL_+ \cap GL_\circ$ and $\gamma \in GL_\circ$, if any one of them is not, then just generate a new random λ or γ and test them again. Something like this can be found in Listing 3.

The checks are made in the creation of the HiddenSum and if it raises a flag that the combination is not vulnerable, we simply try another one. Brute force works well for matrices up to a block size of 5. For a 5×5 matrix the number of combinations is $2^{25} = 33,554,432$. The number of combinations to test scales exponentially, however, so when finding a block size greater or equal to 6 it is advised to generate a λ using the improved $find\_attackable\_lambda$ function which has to be used manually. How this function work will be explained in detail later.

The save functionality in the cipher is rather straightforward. Either the user can create the file and load it into the toy cipher class, or a cipher can be created within the framework and saved to a file. The save file looks like in Listing 4, which is a 3×3 vulnerable γ and λ combination.

```
from ToyCipher.ToyCipher import ToyCipher
from Attack.HiddenSum import HiddenSum

key = "101"
t = ToyCipher(block_len=3, rounds=5)
t.load_cipher("vulnerable_S_and_P.txt")
hs1 = HiddenSum(N=3, k=1, t=t, key=key)
hs2 = HiddenSum(N=3, k=1, t=t)

c = t.encrypt("010", key)

m1 = hs1.attack(c)
m2 = hs2.attack(c, key)

print("{} {}".format(m1, m2))
```

Listing 2: Create a 3×3 ToyCipher and decrypt an encrypted message using the HiddenSum class.

4.7 Improve Finding Vulnerable λ

Since the size of λ increases exponentially, we have to apply some way of finding these matrices more efficiently. Algorithm 4.2.1 in Carlo Brunetta's master thesis has a solution to this [7]. What this algorithm points out is that if λ is constructed in a very specific way and $\lambda \in GL_+$, then we also know that $\lambda \in GL_\circ$, thus it is vulnerable.

$$\lambda = \begin{bmatrix} A & D \\ C & B \end{bmatrix} \qquad \begin{cases} A \in GL_+(n) \\ B \in GL_+(k) \\ C \in 0^{k \times n} \\ D \in Rand_{0,1}(F^{n \times k}) \end{cases}$$

This is a significant reduction in the number of matrices that we have to check. For example, if we want to find a 5×5 matrix that is vulnerable before we would have to test $2^{25} \approx 3.35 * 10^7$ matrices. With the algorithm, we instead can focus on A and B. By setting $A \in F^{3\times3}$ and $B \in F^{2\times2}$, we only need to find a combination where A and B is in GL_+. We basically can reduce the entirety of C since we know that it only is zeros in dimension $k \times n$.

In case we brute force to find a λ now, we have reduced the number of matrices from 2^{25} to $2^9 2^4 2^6 = 524288$ different matrices, this is a reduction of $\approx 98.4\%$. A graphical representation where all the ones at a specific position of a vulnerable λ have been added together can be observed in Figure 4.3. As we can observe, the bottom left area does not consist of any matches, so we can safely ignore this area when performing a brute force search, also worth noting is that it is k and the identity that decide this area of zeros. Using the improved way of discovering λ

```
from ToyCipher.ToyCipher import ToyCipher
from Attack.HiddenSum import HiddenSum

while True:
    try:
        t = ToyCipher(block_len=3, rounds=1)
        hs = HiddenSum(N=3, k=1, t=t)
        break
    except ValueError:
        continue
t.save_cipher("found_vulnerable.txt", hard=True)
```

Listing 3: Find a vulnerable cipher and save it using brute force.

with this property an example could be done like shown in Listing 5.

Because of the reduction in the number of matrices we have to test, it gives us an obvious improvement in speed! If we increase the block size from 5 to 6, the improved function finds a matrix in a few seconds. The old brute force version however we have had running for 2 days without finding a single match. The reason for this can quite easily be calculated. In the case of finding a vulnerable λ' where $N = 6$ we have to choose a $k = 3$, hence we know the size and position of the zero area.

$$\lambda' = \begin{bmatrix} x_{0,0} & x_{0,1} & x_{0,2} & x_{0,3} & x_{0,4} & x_{0,5} \\ x_{1,0} & x_{1,1} & x_{1,2} & x_{1,3} & x_{1,4} & x_{1,5} \\ x_{2,0} & x_{2,1} & x_{2,2} & x_{2,3} & x_{2,4} & x_{2,5} \\ \boxed{0} & \boxed{0} & \boxed{0} & x_{3,3} & x_{3,4} & x_{3,5} \\ \boxed{0} & \boxed{0} & \boxed{0} & x_{4,3} & x_{4,4} & x_{4,5} \\ \boxed{0} & \boxed{0} & \boxed{0} & x_{5,3} & x_{5,4} & x_{5,5} \end{bmatrix}$$

Assuming that Python's random function perfectly has a 50% probability to either choose a 0 or 1, we hence have to get 9 zeros in 9 very specific places. The probability to get this alone is $0.5^9 \approx 0.00195 = 0.195\%$. Even though this alone is a very small probability, it is only the probability for us to get a shot of randomly generating two other $GL_+^{3\times 3}$ matrices and a randomly generated 3×3 matrix that make the whole matrix linear. Hence, the probability of this happening is very unlikely.

4.8 γ Only Partially $\circ$-linear

The exploration of γ has been quite elementary up to this point. All that has been made sure of is that $\gamma \in GL_\circ$ and if that is the case, the attack has to work combined with a $\lambda \in GL_+ \cap GL_\circ$. However, something yet unexplored is if the same attack will work if we only have γ that is partial $\circ$-linear. For a γ to be possible to be in $GL_\circ$ it must be of a block size bigger or equal to three, so for a minimalistic example, we

```
## P BOX
1 1 0
0 1 0
0 0 1

## S BOX 0
0 0
1 7
2 3
3 1
4 5
5 6
6 2
7 4

## K BOX
0 5
1 6
2 7
3 1
4 4
5 2
6 0
7 3
```

Listing 4: How a saved cipher file looks like, in case of multiple γ each should receive an individual identification number (in this example we have one and it is zero).

```
from ToyCipher.ToyCipher import ToyCipher
from Attack.HiddenSum import HiddenSum

def find_l(N, k):
    hs = HiddenSum(N=N, k=k)
    t = ToyCipher(block_len=N)
    P, P_I = [], []
    while True:
        P, P_I = t.find_attackable_lambda(N=N, k=k)
        if hs.lambda_GL_ring(P) != [] and hs.lambda_check(P=P):
            break
    print(P)

find_l()
```

Listing 5: Find vulnerable λ using the new and improved way.

have to find a compatible λ with a block size of six to explore this further.

To focus on the point that is if splitting γ into sections will work with the current implementation, a λ has been chosen that is equal to the identity matrix. This completely defeats the purpose of having a mixing layer at all since $I = I^{-1}$, but at least by doing this we can see if the whole implementation works as expected with divided up γ.

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

$$\gamma_0 = \{0:0, 1:4, 2:2, 3:1, 4:3, 5:7, 6:5, 7:6\}$$

$$\gamma_1 = \{0:0, 1:6, 2:7, 3:3, 4:2, 5:4, 6:1, 7:5\}$$

All the rules are accurate so we create the ToyCipher and create the HiddenSum attack.

However, even with λ being the identity matrix, thus only relying on the confusion property from the two γ-boxes, the attack does not work. This, at its core, makes sense. The attack is built as a combination of the whole λ and γ, thus when combined the $\circ$ property of the γ parts do not combine with the $\circ$ property of λ. If the *combination* of γ_0 and γ_1 is $\circ$-linear however, the attack would still work. But in this case, only each sub-section of γ is $\circ$-linear, so the combination breaks the linearity and we would have to come up with a way to maintain the sub-sections linearity throughout the blockcipher.

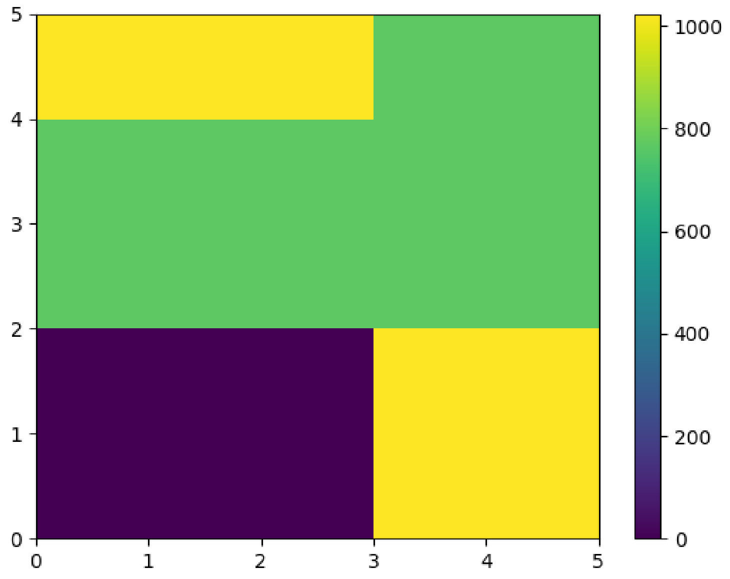


Figure 4.3: Representing a 5×5 matrix where we have brute forced every possible combination and summed up the 1's on each position for every vulnerable cipher that holds the $\lambda \in GL_+ \cap GL_\circ$ property with $k = 2$. Can hence see that the lowest left 2×3 box always have to be zero for the property to hold.

```python
from ToyCipher.ToyCipher import ToyCipher
from Attack.HiddenSum import HiddenSum

t = ToyCipher(block_len=6, rounds=1, num_of_gamma=2)
t.load_cipher('./saved_cipher.txt')

hs = HiddenSum(N=6, k=3, t=t)

for i in range(2**6):
    c = t.encrypt("101010", t.int_to_binary(i, 6))
    m = hs.attack(c, k=t.int_to_binary(i, 6))

    if t.binary_to_int("101010") != m:
        print("Not equal")
```

Listing 6: Test attacking using separate γ and an identity λ.

5
Discussion

The overall development of the framework has been successful. The hidden sum algorithm has been implemented and can be used and deployed dynamically. When finding a cipher it is also possible to save it and later load it for easy access. Everything was not successfully implemented however and is worth being further explored, especially to calculate the hidden sum when using multiple S-boxes are used.

5.1 Structure of the Framework

The framework has been developed to be scalable and easy to use. For this to be achieved, we have spent time to try and divide each operation into the class it fits best, both to be able to inherit the classes during development and for the usability of the framework. The efficiency of the framework is as previously discussed not optimal because Python is used instead of Cython.

The attack itself is built to create a summation that is equivalent to decryption with a specific key, hence removing all the security benefits that rounds give.

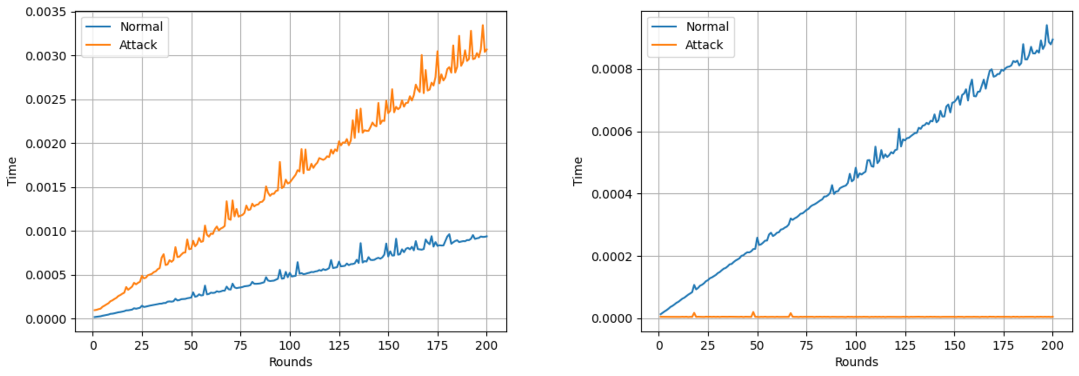


Figure 5.1: (Left) The decryption of a cipher with a new key in the attack and have to reconstruct the attack every time from 1 to 200 rounds. (Right) The decryption of a cipher using a specific key from 1 to 200 rounds created initially.

Figure 5.1 represents two ways of using the HiddenSum framework. The left picture reconstructs the hidden sum for each selection of rounds and as we can see this is much slower than simply decrypting using the key. This is however not something concerning but instead something that would be expected. To construct R for a specific key there have to be as many encryptions as the dimension of the block, plus the zero bases. However, the strength of the hidden sum can be seen in the right picture, when R is constructed we will be able to calculate its value in constant time, no matter the number of rounds in the cipher.

5.2 Finding λ that is Vulnerable

As we have explained before in Section 4.7, the search for a vulnerable λ can be improved. In the framework, the difference between finding a vulnerable λ using the new implementation and randomly generating a matrix can be compared in Figure 5.2. Going any bigger than this, the old randomly generated matrices takes so much time to be able to find it would make spike from 4 to 5 unrecognizable in the graph while the new implementation still can find one in a matter of seconds.

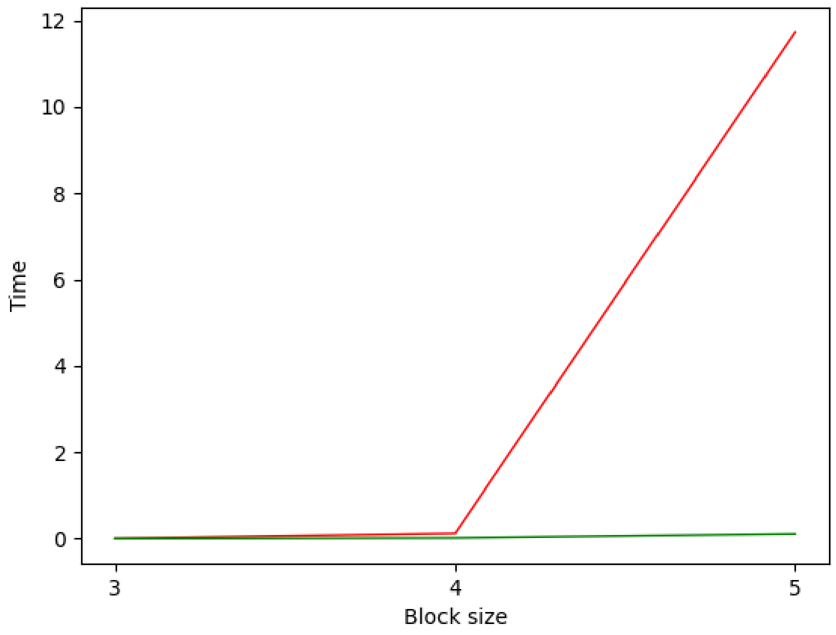


Figure 5.2: (Red) Using randomly generated matrices. (Green) Using the new and improved implementation.

Not using this from the start was a mistake. A lot of time was spent on constructing vulnerable λ and γ, if we were to use the optimized algorithm for finding λ at the start, more time could have been spent on figuring out a clever way of finding γ of interest. More time should have been spent on the theory surrounding the creation of the matrices than was made.

5.3 Conclusion

During the thesis process, the main goal of developing the framework has been achieved. The framework is working and can be used to create the hidden sum vulnerability for any block size as long as the rules of λ and γ are upheld. The framework hopefully will be useful for both research and education for people wanting to see a practical application of a trapdoor in a blockcipher. Algorithms that improve the efficiency of finding vulnerable λ are also implemented within the framework alongside a saving and load functionality.

Splitting γ into multiple sub-sections is however still unsolved but can be used within the framework. To get this to work, a new way of structuring λ will probably have to be figured out to maintain the $\circ$ property across encryption and decryption. Whether it is possible or not is however still unknown. More research surrounding this is thus needed.

Bibliography

[1] R. F. Churchhouse, "Codes and Ciphers: Julius Caesar, the Enigma, and the Internet." Cambridge: Cambridge University Press, 2002. Internet resource.

[2] J. F. Dooley, "History of Cryptography and Cryptanalysis: Codes, Ciphers, and Their Algorithms." Cham, Switzerland: Springer, 2018. ISBN 9783319904429.

[3] H. Henderson, "Alan Turing : computing genius and wartime code breaker" New York, NY : Chelsea House, 2011. ISBN 9780816061754.

[4] E. Biham, A. Shamir, "Differential Cryptanalysis of the Data Encryption Standard" Springer, 1993. ISBN 9781461393160.

[5] R. Morelli, "The Vigenere Cipher." Trinity College, www.cs.trincoll.edu/ crypto/historical/vigenere.html. Accessed 18 Sept. 2017.

[6] R. Simpson, "Cryptology, the Secret Battlefield of World War I: Dawn of the Crypto Arms Race", CSUCI history department, guest lecture. 2018, accessed online.

[7] C. Brunetta, "On some computational aspects for hidden sums in boolean functions", Master Thesis, University of Trento, 2015-2016.

[8] C. E. Shannon, "Communication theory of secrecy systems", The Bell System Technical Journal, vol. 28, no. 4, pp. 656-715, 1949.

[9] The General Public License Version 3 (GPLv3), website: https://www.gnu.org/licenses/gpl-3.0.en.html [Accessed: 27 Jan 2022]

[10] M. Calderini, "On boolean functions, symmetric cryptography and algebraic coding theory", Ph.D. thesis, University of Trento - Department of Mathematics, April 2015.

[11] L. R. Knudsen, M. Robshawr, "The Block Cipher Companion", 1st ed. 2011, Springer Berlin Heidelberg, ISBN 9783642173424, 2011.

[12] R. Pindyck, D. Rubinfeld, "Linear Algebra and Its Applications", Global Edition, ISBN 9781292081984, 2015.

[13] L. Rompis, "Boolean Algebra for Xor-Gates". Journal of Computer Sciences and Applications, 1(1), 14-16, 2013.

[14] A. Caranti, F. Dalla, M. Sala, "Abelian regular subgroups of the affine group and radical rings", arXiv preprint math/0510166, 2005.

[15] C. H. Li, "The Finite Primitive Permutation Groups Containing an Abelian Regular Subgroup", Proceedings of the London Mathematical Society, 87(03), 725–747, 2003.

[16] L. R. Knudsen, "Contemporary Block Ciphers", In: Damgård I.B. (eds) Lectures on Data Security. EEF School 1998. Lecture Notes in Computer Science, vol 1561. Springer, Berlin, Heidelberg, 1999.

[17] J. Ma, Y. Li, "Gauss–Jordan elimination method for computing all types of generalized inverses related to the 1-inverse". Journal of Computational and Applied Mathematics, 321, 26–43, 2017.

[18] N. Pramstaller, F. K. Gurkaynak, S. Haene, H. Kaeslin, N. Felber, W. Fichtner, "Towards an AES crypto-chip resistant to differential power analysis". In Solid-State Circuits Conference, 2004.

[19] E. Biham, & A. Shamir, "Differential cryptanalysis of DES-like cryptosystems". Journal of Cryptology, 4(1), 3–72, 1991.

[20] S. Chatterjee, H. Nath Saha, A. Kar, A. Banerjee, A. Mukherjee, & S. Syamal, "Generalised Differential Cryptanalysis Check for Block Ciphers". 2019 IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON), 2019.

[21] D. Wagner, "The boomerang attack". In International Workshop on Fast Software Encryption (pp. 156-170). Springer, Berlin, Heidelberg, 1999.

[22] L. Knudsen, “DEAL-a 128-bit block cipher”, Complexity, vol. 258, no. 2, 1998.

[23] L. Steven, "Crypto: How the Code Rebels Beat the Government — Saving Privacy in the Digital Age". Penguin Books. pp. 55–56. ISBN 0-14-024432-8, 2001.

[24] A. Tardy-Corfdir & H. Gilbert, “A known plaintext attack of FEAL-4 and FEAL-6”, Proceedings of Crypto’ 91, 1991.

A

Code Implementation

A.1 Calculate inverse using XOR

```python
def calculate_inverse(self, A_t):
    A = copy.deepcopy(A_t)
    I = self.get_identity(len(A))
    foundPivot = False
    for i in range(len(A)):
        if A[i][i] != 1:
            for j in range(i+1, len(A)):
                if A[j][i] == 1:
                    foundPivot = True
                    A = self.row_shift(A, i, j)
                    I = self.row_shift(I, i, j)
                    break
            if foundPivot == False:
                return []
            else:
                foundPivot = False
        for j in range(i+1, len(A)):
            if A[j][i] == 1:
                A = self.matrix_mul_row(A, i, j)
                I = self.matrix_mul_row(I, i, j)

    for i in range(len(A)-1, 0, -1):
        for j in range(i-1, -1, -1):
            if A[j][i] == 1:
                A = self.matrix_mul_row(A, i, j)
                I = self.matrix_mul_row(I, i, j)
    return I
```

A.2 Calculate inverse using the ∘-operator

```
def lambda_GL_ring(self, A_t):
    r = Operations(N=self.N, k=self.k)
    A = copy.deepcopy(A_t)
    I = self.get_identity(len(A))
    foundPivot = False
    for i in range(len(A)):
        if A[i][i] != 1:
            for j in range(i+1, len(A)):
                if A[j][i] == 1:
                    foundPivot = True
                    A = self.row_shift(A, i, j)
                    I = self.row_shift(I, i, j)
                    break
            if foundPivot == False:
                return []
            else:
                foundPivot = False
        for j in range(i+1, len(A)):
            if A[j][i] == 1:
                A_t = copy.deepcopy(A)
                A_t[j] = self.int_to_binary(
                            r.ring(self.binary_to_int(A_t[j]),
                                    self.binary_to_int(A_t[i])),
                                    self.N
                            )
                A = A_t

                I_t = copy.deepcopy(I)
                I_t[j] = self.int_to_binary(
                            r.ring(self.binary_to_int(I_t[j]),
                                    self.binary_to_int(I_t[i])),
                                    self.N
                            )
                I = I_t
    for i in range(len(A)-1, 0, -1):
        for j in range(i-1, -1, -1):
            if A[j][i] == 1:
                A_t = copy.deepcopy(A)
                A_t[j] = self.int_to_binary(
                    r.ring(self.binary_to_int(A_t[j]),
                                self.binary_to_int(A_t[i])),
                                self.N
                            )
                A = A_t

                I_t = copy.deepcopy(I)
                I_t[j] = self.int_to_binary(
                            r.ring(self.binary_to_int(I_t[j]),
                                   self.binary_to_int(I_t[i])),
                                   self.N
                            )
                I = I_t
    return I
```