CYCLE EXPERIMENTS ON THE CRYPTOGRAPHIC PERMUTATION XOODOO

Master's Thesis

GOURI VIRAVALLI

CYCLE EXPERIMENTS ON THE CRYPTOGRAPHIC PERMUTATION XOODOO

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE
Specialization CYBER SECURITY

by

GOURI VIRAVALLI
born in INDIA

CYBER SECURITY
Department of Computer Science
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
http://wis.ewi.tudelft.nl

CYCLE EXPERIMENTS ON THE CRYPTOGRAPHIC PERMUTATION XOODOO

Author: GOURI VIRAVALLI
Student id: 4738888
Email: gouri7@outlook.com

Abstract

Cryptography is the science of concealing messages, and it provides data security and privacy. A cipher is designed to be as secure as possible, to not be easily broken with the current availability of computational power in a reasonable amount of time. Various attacks have been discovered over time, and recently, the invariant subspace attack was presented on the PRINTcipher. This is a statistical saturation attack which makes use of the weak keys of the cipher.

This report delves into the cryptographic permutation known as Xoodoo, and explores the possibilities of its vulnerability towards the invariant subspace attack. For this, we investigate the cycle structure of the permutation by performing cycling experiments on its round function. We implement the naive Xoodoo round function after stripping off its round constants, and take advantage of the symmetry properties of the Xoodoo state to understand how the symmetry classes behave. The identification and description of symmetry classes of Xoodoo states based on the concept of lattices help us observe some of the symmetry classes that are small enough and can fully determine their entire cycle structure. With an exception for one anomaly case, there were no observed deviations from the behaviour of Xoodoo in comparison to the behaviour of a random permutation, and the cycles were found to have no particular structure. We thus conclude that it is highly unlikely that Xoodoo is vulnerable to invariant subspace attacks. Many factors about the algebraic background of the cipher were taken into consideration for the implementation. The parity of the cycle count, the behaviour of the cipher with different symmetry classes with a given state size, the interaction between bits in the intermediate states of the cycles, and the factors that influenced the number of cycles were all analysed. The concept of symmetry is rigorously described. We establish that the number of cycles increases as the length of the input permutation decreases. The outcomes of the experiments are compared with the theory behind the implementation and anomalies are explained.

Thesis Committee:

Chair:	Dr Jan van der Lubbe, Faculty EEMCS, TU Delft
University supervisor:	Dr. Stjepan Picek, Faculty EEMCS, TU Delft
External supervisor:	Dr. Joan Daemen, Faculty of Science, Radboud University
Committee Member:	Dr. Elvin Isufi, Faculty EEMCS, TU Delft

Acknowledgements

I have had such an enriching time working on this thesis. It was stressful, but a good kind of stressful. Since the time I was in school, I've loved mathematics, and later in college I discovered cryptography. My thesis combined the two topics that I am passionate about, and I am ever so grateful to have had this opportunity.

I want to thank my daily supervisor Dr. Stjepan Picek who guided me in the right direction and was always supportive, motivating, and frank. Honestly, he's one of the best teachers I have worked with. I thank my external supervisor Dr. Joan Daemen, who patiently and clearly explained concepts to me that I would probably never understand by just reading. If in 2015 (when I got 4/4 marks for writing the AES pseudocode in an exam) I was told that I would work on my Master's thesis with Joan, there is no way I would believe it. Both my supervisors provided me with great ideas, relevant literature, and most importantly - critical feedback. Furthermore, I thank Dr. Jan van der Lubbe and Dr. Elvin Isufi for being on my thesis committee and taking out time to read my report.

I want to thank my mumma, Gou (mostly for the memes), and the rest of my family in India - nothing would be possible without their love. I owe them so much, I hope they are happy and proud of me. I am also very grateful for all my friends in Delft and India. Special thanks to Gabi for supporting, motivating, and feeding me each time I had a deadline. Last but not the least, thank you Andrej, for bringing home to me. You were my support system, and you believed in me and my thesis more than I did. Half of my grade is yours (joking).

I dedicate this thesis and all my hard work to my dearest thatha - my grandpapa who passed away earlier this year. He was the first professor I knew, the first engineer I saw in action, the oldest techie I had met, the most disciplined person I had seen, and the perfect grandfather. He was a big part of my life and meant so much to me. I miss him every day and love him very much. Below is a picture of a 6 month-old me sitting on my thatha's lap, playing on his computer. It was an old x86. The MS DOS era, of course.

GOURI VIRAVALLI
Delft, Netherlands
August 15, 2019

Contents

List of Figures

List of Tables

Notations

x	x-coordinate
y	y-coordinate
z	z-coordinate
A	A Xoodoo state
A_y	Plane ’y’ of state ’A’
M x N	A plane with coordinates x = M and z = N
$\bar{A}$	Bit compliment operation over A
$A_n \lll (a,b)$	Cyclic shift of plane A_n in directions x and z to positions $(x+a, z+b)$
$\oplus$	Bitwise XOR operation
$\bigodot$	Bitwise AND operation
τ	A transformation on a vector space
$\vert$	Such that
$\in$	Belongs to
$\forall$	For all
$\circ$	Function composition operation
χ	The chi function
θ	The theta function
ρ_{west}	The rho west function
ρ_{east}	The rho east function

Chapter 1

Introduction

Throughout the history of human evolution, one field of constant and major development has been communication. The boom of the Internet and the birth of Industry 3.0 further saw the usage of wireless methods of communication. As wireless communication became easily accessible to more and more people, the amount of data being generated began to increase exponentially. This resulted in the transmission of huge amounts of data with storing capabilities. [24]. Data storage/memory became much more popular, affordable, and powerful compared to conventional storage media such as paper.

At every stage of this technological progress, there was, is, and always will be a necessity to ensure that all this data is kept secure and protected from parties for whom the data was not intended. Security is a required measure, and cryptography provides exactly that. Cryptography is a science of concealing messages, and it can provide the privacy of and security for information and data. This is achieved by manipulating data in such a way that it can only be read by authorized parties [11]. Encryption and decryption are the two most important processes involved in this science. Encryption encodes a message to make it unreadable, and decryption decodes the encrypted message back to the original message. These processes break down the input to its simplest form, and then make extensive use of mathematical theory and sometimes also electrical and communication engineering to achieve this.

Information science is a field that involves the study of information - beginning with how this information came into existence, further on to its storage, analysis, use, manipulation, retrieval, and its protection. Information security is a field encompassed by information science, and this domain broadly involves the protection of information. This can be done either by encrypting the information, or by removing any possible security risk, or both. The three aspects of information science include confidentiality, integrity, and availability of data, all of which directly or indirectly depend on cryptography.

A cipher is an algorithm that is written to perform such encryption or decryption. A plain-text is the message that is to be encrypted, and cipher-text is the output of this algorithm performed on the plain-text. The first cryptographers only used pen and paper, and later they moved on to make use of the ever-growing computing power. The main goal of any cipher is to protect the data and not be broken by an adversary [21]. Hence, today, a cipher's design should be very secure and must take into consideration various computational and technological advances.

The biggest advancements in the field of cryptography in recent times were marked by the creation of Public Key Encryption by Diffie and Hellman [8], and the Rivest Shamir Adleman cryptosystem (RSA) [18]. The Data Encryption Standard (DES) [9] was a widely used cipher, which is now replaced by the Advanced Encryption System (AES)[6]. Some of these are explained further in Chapter 2. Social media applications such as Whatsapp and Facebook make use of the AES encryption. Credit card companies, banks, and the Payment Card Industry in general also follow specific guidelines which include ciphers such as AES. Many browsers, email applications, virtual private networks, etc. make use of RSA.

The world has reached a stage where it is not possible to survive without encryption. Individuals, businesses, corporations, and governments around the world - everyone is heavily reliant on encryption and decryption to keep their data safe. For this, the ciphers that are in use must be constructed in such a way that they cannot be easily broken with the available computational power and a reasonable amount of time.

1.1 Motivation and objectives

If a cipher that is in use is broken, then it no longer fulfills it's purpose and should be removed from its application. There are many methods to check if a given cipher is secure or not. During the process of the cipher design itself, the authors of the cipher perform tests to ensure security, thus many of the basic security risks or flaws are analyzed at the start. But, with time and increasing computational power, there might be undiscovered gaps in the cipher. Cryptanalysis, as the name explains, is a method of attacking a cipher by analyzing the given cipher-text to gain as much information as possible about the plain-text.

This report analyses the cryptographic permutation known as Xoodoo, designed by Daemen et al. [7]. Xoodoo was chosen as the topic of choice for this thesis as it is an amalgam of two other cryptographic techniques with various desirable properties. It has excellent propagation properties, and is a lightweight permutation that is efficient on low-end processors. The main application of Xoodoo is in the Farfalle construction, and it allows building authenticated encryption schemes and MAC functions. It also allows a high level of parallelism. The design of Xoodoo is explained in Chapters 2 and 3. Xoodoo takes as input 384 bits and outputs a permuted version of the input. The permutation involves no key and has a round function that is made of a series of five steps that alter the bits of the input. The objective of this thesis is to perform cyclic experiments on Xoodoo using varying inputs, to understand the permutation and its vulnerability to the invariant subspace attack.

This is particularly interesting because as far as we are aware, there has been minimal work regarding the cyclic behavior of ciphers and what they indicate. This is a topic that has not been explored in detail before, thus any information gained from this research would either be backed by theory, or has to be disproved by experimentation. Related research on such cycling experiments include a paper by Kaliski et al. [10]. wherein the cyclic structure of DES was analyzed. This paper is further discussed in detail in Chapter 3. There has also been no prior work performed on the cycles of Xoodoo, which makes it even more interesting to see what we can conclude from the results.

1.2 Research statement

The primary goal of this thesis is to investigate whether the cryptographic permutation Xoodoo is vulnerable to invariant subspace attacks. An invariant subset is one where the elements of a set remain unchanged when a transformation is applied to it. Consider f to be a mapping on a given set U of elements, and let S be a subset of U. Let f be a self-map such that:

$$f : S \to S$$

Now, the subset S is said to be invariant under f if and only if the following condition is met [31].

$$\forall s \in S : f(s) \in S$$

Now, to understand an invariant subspace attack, let us consider a subset S of the domain of the permutation. Let F be the permutation on the elements of the subset [30]. The subset S is an invariant subspace of the permutation F if:

$$\{F(A) | A \in S\} = S$$

Invariant subspace attacks take advantage of the fact that the round function maps an input from a subspace of the state space to the same subspace. This is a weakness as it drastically reduces the output space. Invariant subspace attacks are a threat to lightweight block ciphers with one of two design specifications[14]:

- There is no key scheduling algorithm that modifies the key for each round, or rather the key remains the same for all rounds of the round function. If round keys are present, then they consist of the cipher key and a round-specific constant. This round constant eliminates the symmetry present in the other steps of the round function.

- The property of symmetry is destroyed by adding round constants in a separate step irrespective of the key.

Xoodoo being a cryptographic permutation has neither a cipher key nor a round key, but the round function of Xoodoo involves a step mapping that adds round constants to destroy symmetry. The concept of symmetry is explained further in Chapter 4. For this research, we consider the round function of Xoodoo but strip off the step that adds round constants to the input. We refer to this as the *naive round function*, and it is formally defined in Chapter 2. The naive round function now exhibits symmetry and thus will lead to invariant subspaces. It can be defined for any dimension of Xoodoo, and each dimension defines a different permutation.

For its domain, every permutation will form cycles. If the F that was considered above was a permutation, then every cycle that is formed by F, is such an invariant subset. But, invariant subspace attacks only become feasible if there are very many cycles. This is possible if F is very far from being a random permutation. Another possibility that gives way for an invariant subspace attack is if the cycles have a particularly recognizable structure. A random permutation is a random ordering of a set of objects, and a cycle structure close to that of a random permutation is evidence that an invariant subspace attack is very likely not possible.

The bare Xoodoo round function defines a permutation over a set of elements that maps members of its set to the same set. This set is called as a symmetry class, and it forms a partition of the state space. A formal definition of the same can be found in Chapter 4. In this thesis, we investigate the naive Xoodoo round function to examine the invariant subspaces of the permutation and to understand the cyclic structure. If any subspace or symmetry class displays a strong deviation from that of a random permutation, then further investigation would be necessary due to the risk of the invariant subspace attack. We also rigorously describe the property of symmetry and its role in this research.

The presented research attempts to understand Xoodoo by performing experiments on specific symmetry classes of Xoodoo and to investigate the cyclic behavior of the same. Based on this, the following research questions also arise and are answered:

- Is Xoodoo vulnerable to the invariant subspace attack?
- How many cycles do different inputs produce, how and why is it different from what is expected, if at all?
- What are the parities of the cycle structures in the different symmetry classes and how is it so?
- How do the different steps of the round function behave with varying state sizes?
- Describe the role that symmetry plays in this research.
- How do different symmetry classes vary in terms of the cycle count?

1.3 Approach

This investigation is performed by diving deep into the construction of Xoodoo, and further implementing the naive Xoodoo round function. The property of symmetry that the steps of the naive Xoodoo round function exhibit allow the analysis and results of the symmetry classes of Xoodoo with smaller dimensions to be relevant for those with larger dimensions. Each symmetry class displays interesting properties, and out of the possible 37 symmetry classes, 19 have been considered and implemented. Upon implementation of the naive Xoodoo round function, various experiments were performed and the results have been analyzed. Few contributions that have been reported in this thesis are as follows: Identification of how the bit interaction becomes weaker or stronger, and the factors behind it. The parity of the cycle structure, and reasoning for the same. The possibilities for the total number of states involved all cycles put together, and the theoretical explanation behind the numbers that were attained from the experiments.

1.4 Overview of the thesis

Chapter 2 provides background information that is required for following the thesis. The chapter introduces block ciphers and its design elements, and explains the design of the Xoodoo permutation in detail. Chapter 3 is a literature survey of a few identified existing papers and reports on this topic, or topics surrounding the thesis. Chapter 4 describes symmetry in detail, and delves into the various properties of symmetry. It describes various concepts that contribute to the implementation of the naive Xoodoo round function, and they are mentioned through the course of this report. The entire methodology that has been adapted for this research has been explained in Chapter 5. It also expands on the implementation of the naive Xoodoo round function. The results of the experiments are fabricated into Chapter 6, and Chapter 7 advances into the observations that were made from these results. The final chapter evaluates the conclusions that were derived and also mentions some possible future work that this project can be extended to.

Chapter 2

Background

In this chapter, we provide a quick introduction to cryptography, and then provide information on an important method of encryption, namely block ciphers. We explain the concepts of confusion and diffusion that were introduced by Shannon, and then speak about permutation boxes, substitution boxes, and substitution-permutation networks. An overview of Xoodoo and its design specifications are explained, and the steps of the Xoodoo round function are explained in detail with an example. The naive Xoodoo round function is defined and described along with reasoning for its use.

Cryptography is a study that encompasses various techniques to manipulate data to encrypt it. Modern cryptography is heavily reliant on mathematics. Concepts such as factorization, discrete logarithm, elliptic curves, linear algebra, logic gates., etc are used intensively to manipulate or "encrypt" the input. The input that a cipher takes could be of any recognizable form - letters, words, numbers, ASCII values, bits, bytes, and so on. If the cipher is designed to manipulate bits, then these bits could be fed to the cipher in two different ways. These are namely stream ciphers and block ciphers. As the name suggests, stream ciphers requSre a stream of bits as the input and output for encryption. A block cipher, on the other hand, recognizes a set of bits as a 'block', and these blocks are of a fixed length. The following sections expand on block ciphers for two reasons. The first reason is that Xoodoo is permuted and manipulated in blocks. Second, Xoodoo is an iterated permutation, and an iterated permutation can be thought of as an iterated block cipher with a fixed key.

2.1 Block ciphers

Block ciphers are among the most important cryptographic primitives. DES and AES were two groundbreaking block ciphers, which further paved the way to huge amounts of research on the design and analysis of these ciphers [13].

A block cipher is a reversible function that generally has an encryption algorithm E and a decryption algorithm D. Both require the same key k, hence making it a symmetric-key algorithm. A symmetric-key algorithm uses the same key for both encryption and decryption, as opposed to an asymmetric-key algorithm that uses different keys for encryption and decryption, one of which is generally kept private while the other is public. A basic model of an encryption and decryption scheme is seen in Figure 2.1. Block ciphers are designed to be deterministic algorithms, meaning that a specific input and key pair always produces the same output. This is because it goes through the same conditions and sequence of steps each time, and there are no pseudo-random generators or any other factors involved in the process that could cause a probabilistic output.

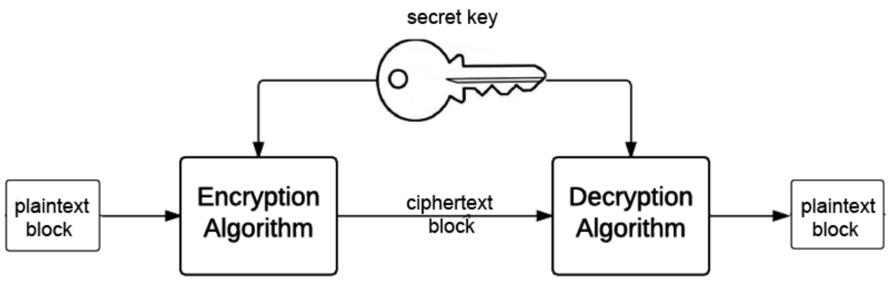

Figure 2.1: An encryption scheme

NIST or the National Institute of Standards and Technology, an American non-regulatory body selected Rijndael to be the encryption standard to be used in various industries in the year 2001. It also falls under FIPS compliant ciphers - FIPS or Federal Information Processing Standard being the standard that most businesses and industries must comply with. Rijndael or the Advanced Encryption Standard (AES) was developed by Vincent Rijmen and Joan Daemen [6] based on the principle of a substitution-permutation network, which is explained in detail later in this Chapter. It can have varying inputs and is a symmetric block cipher.

Before AES, NIST (which was earlier NBS or National Bureau of Standards) was using DES [9] or the Data Encryption Standard, which was designed by IBM. It was also published as a Federal Information Processing Standard but was later withdrawn in 2005 [16]. DES was broken several times in the 1990s[1], and thus there was a need to change the standard. DES had only a 64-bit input size and a 56-bit key size. It was weak against brute force attacks and was broken several times in the 1990s, the fastest being in 1999, in a total of 22 hours [1]. DES is designed based on the principle of a substitution-permutation network.

Block Ciphers can also be used in other cryptographic protocols such as hash functions. Hash functions map some given data of arbitrary size to data of a given size. Another area in which block ciphers can be implemented is in pseudo-random number generators. These are algorithms that generate sequences of numbers, which is in close approximation to sequences of random numbers. PRNGs are also used commonly in cryptography.

In a block cipher, the block of bits undergo various transformations such that the resulting block is always more secure. The most common transformations that block ciphers enforce are substitutions, permutations, logical operations, and modular arithmetic. Though each of these transformations was originally used in different ciphers as the main and only operation, a combination of the lot is now preferred to make the algorithm extra secure. Substitution boxes and permutation boxes can be imagined as white-boxes that take in a specific input and provide a deterministic output for the same, and generally is a manipulation of the bits in the input block. Before delving into substitution and permutation boxes, there are two more crucial concepts to understand.

2.1.1 Shannon's confusion and diffusion

S-boxes and P-boxes are used to make the relation between the plaintext and the ciphertext difficult to understand. Communication Theory of Secrecy Systems [22] is a paper authored by Claude Shannon in 1949, and it introduces two concepts namely confusion and diffusion. They are two important properties for designing a secure cipher. Confusion makes the ciphertext as faint and vague as possible. It also ensures that there is no visible relationship between the secret key and the ciphertext - this is masked by the confusion. Even if a single bit of the key is changed, the whole output would have changed. This property is known as the avalanche effect. Thus even if a hacker has a large number of plaintexts and ciphertexts, it should still be almost impossible to find the key from the information known. This can be achieved if each bit of the ciphertext depends on not just a part of the key, but the entire key, and also in different ways on different bits of the key.

Diffusion is a concept used in block ciphers such that the outputs depend on the inputs in a very complex way. High diffusion can be accomplished with a high avalanche effect, wherein a single bit of the input is changed, then the ciphertext is changed completely, thus spreading the influence of the plaintext over the ciphertext. The relationship between the plaintext and the ciphertext is hence be masked by diffusion.

The best way to achieve confusion and diffusion is to create what is called a substitution-permutation network. This is explained in the next section.

2.1.2 Substitution ciphers and S-boxes

Substitution is a method that replaces each of the elements of an input to the system with another preset element. It takes as input an m bit string and outputs an n bit string which is not necessarily the same size as the input. If the input and the output are of the same size n, then there can be $2^n - n!$ number of S-boxes that can perform this mapping without mapping two different inputs to the same output. S-boxes or substitution boxes are used in symmetric-key algorithms and are mapping tables or lookup tables that map each possible input to an output. These tables can either be kept fixed, meaning they remain constant for every possible input-key pair thus making it a deterministic algorithm, but it can also be generated dynamically from the key. An example of such a cipher that creates a dynamic look-up table is in the Blowfish cipher [20]. Though Blowfish is a widely used and fairly secure algorithm that has not been effectively broken, the Advanced Encryption Standard (or AES) is now preferred. The AES S-box is fixed, unlike Blowfish.

2.1.3 Permutation ciphers and P-boxes

A permutation box or a P-box is used to permute and transpose bits from one form to another. P-boxes are typically classified as compression, expansion, and straight, according to the number of output bits is less than, greater than, or equal to the number of input bits. Although there can be three types of P-boxes, only the straight P-boxes can be inverted. P-boxes are used in cryptography to ensure the shuffling of bits and is used to permute or transpose bits across the S-box inputs. Consider a bit string of length n bits. A P-box would map each of these bits to another bit within the same string. Hence, there can be a total of $n!$ P-boxes that can map an input of n bits to itself.

2.1.4 Substitution-Permutation networks

A substitution-permutation network as understood from the name itself applies various stages of S-box and P-box operations to an input block of a given size. It uses a secret key K, and is constructed with a permutation stage and a substitution stage that is applied to the input iteratively, or in rounds. This SPN model has formed the basis of many private key cryptosystems in the recent past. The Feistel cipher was one of the earliest SPN models. It made use of very similar operations for encryption and decryption, thus removing the requirement of having an inverse permutation or an inverse substitution box. Commonly used ciphers such as DES and AES were also constructed using S-P networks. DES made use of the Feistel structure, while AES did not. A high level of confusion and diffusion is attempted to be maintained to make the cipher more secure. In these systems, the plaintext and the key often have a very similar role in producing the output, hence it is the same mechanism that ensures both diffusion and confusion.

2.1.5 Round functions

In an SP network or a block cipher in general, operations are applied to the input not once, but repeatedly. Making use of the same set of operations iteratively will make the cipher stronger against cryptanalysis. The single set of fixed operations that are repeatedly applied to the input is collectively called the round function of the cipher. It can consist of many operations or steps, and the number of rounds represents the number of times the round function is executed on an input for encryption, or an output for decryption.

2.2 Xoodoo

Xoodoo is a cryptographic permutation-based cipher designed by Joan Daemen, Seth Hoffert, Gilles Van Assche, and Ronny Van Keer. The design of Xoodoo was inspired by Keccak-p [3]. Keccak is a cryptographic primitive based on the sponge construction, and it operates on a fixed-length permutation. The structure of Xoodoo was taken from Gimli [2], which is a permutation function designed to build high security and high performance ciphers that are energy efficient. Xoodoo's primary target application is in the Farfalle construction. Farfalle [4] is a mode of use based permutation, which means that it can be used in different modes depending on the use case. The different modes could be for encryption, as a MAC, etc. This entire construction is designed to achieve efficiency for low-end processors by providing a high level of parallelism. Keccak, Gimli, and Farfalle are all discussed in detail in Chapter 3. Apart from Farfalle, Xoodoo can also be used in other permutation-based modes. It is found that Xoodoo has excellent differential propagation and correlation properties.

2.2.1 Design and specifications

The Xoodoo permutation operates on the space of 384-bit strings. A round function consisting of five steps is applied to the input iteratively for a pre-specified number of rounds. The input is viewed as a three-dimensional array, and this structure is called a state. A state consists of 384 bits and can be visualized as three horizontal planes one above the other. Each of these planes is made up of lanes. The width of the input represents the number of lanes, ranging from 1 to 4 in powers of 2. Thus there can be $1,2$, or 4 lanes. These bit positions in the state are represented by the x-coordinate, which is the coordinate along the horizontal axis, and it ranges from 0 to $width-1$. Each lane also has a length that ranges from 1 to 32, again in powers of 2, and hence can take the values 1, 2, 4, 8, 16, and 32. The bits here represented by the z-coordinate, and we can say that z ranges from 0 to $length-1$. A state is made up of columns when viewed from a top-down approach. The height of the state is also the size of a column, and this is always a constant value of 3. The bits of a column are represented by the y-coordinate along the vertical axis and ranges from 0 to 2.

A Xoodoo state has the following dimensions:

- width = 4
- length = 32
- height = 3

A state can be broken down into planes, sheets, lanes and columns. Figures 2.2 and 2.3 illustrate these parts in a small version of Xoodoo with length = 8 bits instead of 32 bits, and each of them are explained below.

- A state is three-dimensional and has dimensions 4x8x3.
- A plane is a two-dimensional part in the $x-z$ directions that has a width of 4 bits and length 8 bits. There are 3 planes in a state.
- A sheet is again a two-dimensional part, but in the $y-z$ direction, and has a length of 8 bits and a height of 3 bits. A Xoodoo state is made up of 4 sheets.
- A lane is a one-dimensional part of the state in the z-direction, it is made up of 8 bits, and there can be 12 lanes in a 96-bit input state.

- A column is made up of three bits, one bit from each plane. It is one-dimensional in the y direction. There are 32 columns in a state.

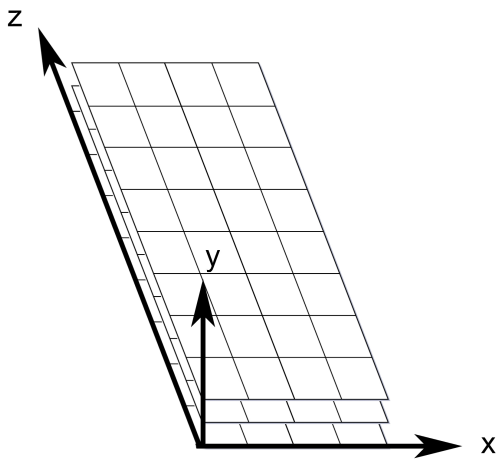

Figure 2.2: A Xoodoo state

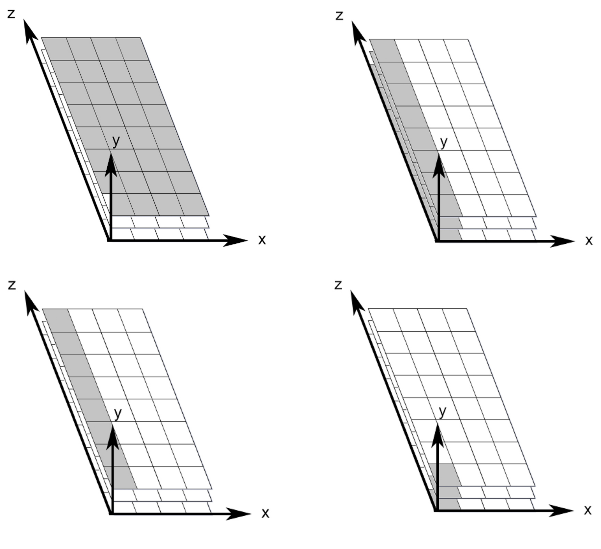

Figure 2.3: Clockwise from top: Xoodoo plane, Xoodoo sheet, Xoodoo lane, Xoodoo column

An input of 384 bits is initially stored as a string s. If the bits of this string is indexed by i, then the position of the bit on the state with respect to the position of the bit in the input string is calculated as follows.

$$i = z + 32(x + 4y) \tag{2.1}$$

2.2.2 Xoodoo's round function

As explained in the previous section, Xoodoo is a cryptographic permutation where a round function operates over a given input for a specific number of iterations. Five step mappings constitute this round function. The input is first sent to the mixing layer θ, followed by a plane shifting layer ρ_{west}, the addition of round constants, a non-linear layer χ, and finally, a second plane shifting layer ρ_{east}. The algorithm of the round function R for a given number of rounds n_r on a state A is shown below. [7] defines the permutation Xoodoo for a total of 12 rounds, making $n_r = 12$. The index of the rounds begins at $1 - n_r$ and increases until it becomes 0. The numbering is designed so to avoid slide attacks. A slide attack is a form of cryptanalysis that makes a high number of rounds of the round function irrelevant to the security of the cipher. This type of attack analyses the key scheduling of the cipher, and exploits a weakness in the same, and generally when the keys are cyclically repetitive [5].

Algorithm 1 The round function of Xoodoo

```
Require: $n_r$ = number of rounds, $i$ = round index, $A$ = state of Xoodoo
  for $i$ from $1 - n_r$ to 0 do
    A ← MixingLayer(A)
    A ← RhoWest(A)
    A ← AdditionOfRoundConstants(A)
    A ← ChiFunction(A)
    A ← RhoEast(A)
  end for=0
```

Each of the five step mappings of the round function is explained in detail with an example in the following sections. The example input that we use to explain the round function is 96 bits in size. It has three planes with a width of 4 bits and a length of 8 bits each.

The mixing layer

The mixing layer (also written as θ) is the first step mapping in the round function and constitutes three sub-steps. The first of these sub-steps within the mixing layer is to create a parity plane. This is calculated by calculating the bit-wise sum of the three planes. A mixing effect or a θ-*effect* is computed by performing a bit-wise addition of two shifted versions of the parity plane that was calculated in the previous step. The θ-*effect* is then added to each of the three original planes, again in a bit-wise manner. This makes the θ layer a column parity mixer. The algorithm for the θ function is as follows:

$$\begin{aligned}&P \leftarrow A_0 + A_1 + A_2\\&E \leftarrow P \lll (1,5) + P \lll (1,14)\\&A_y \leftarrow A_y + E, y \in \{0,1,2\}\end{aligned}$$

An example of how the mixing layer acts on a state is shown in Figure 2.4 below.

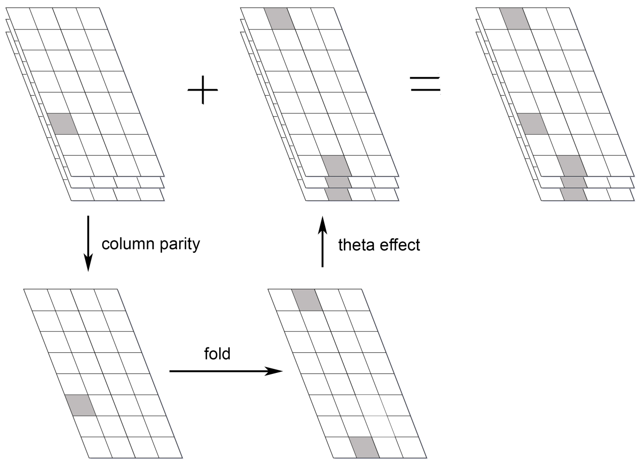

Figure 2.4: An example of the θ function operating on a single bit state

The non-linear function

The non-linear layer or the χ function is the fourth step mapping of the round function. It operates in parallel on the 3 bit columns of a state, and forms a layer of 3 bit S-boxes. If the input is of size 96 bits, then there would be a total of 32 S-boxes. It is evident that the interaction between the bits only occurs among the columns, and not within a plane. The algorithm for the χ function is given below as:

$$
\begin{aligned}
&B_0 \leftarrow \bar{A_1}.A_2\\
&B_1 \leftarrow \bar{A_2}.A_0\\
&B_2 \leftarrow \bar{A_0}.A_1\\
&A_y = A_y + B_y \forall y \in \{0,1,2\}
\end{aligned}
$$

An illustration of the effect of the χ function on a state is shown in Figure 2.5 below.

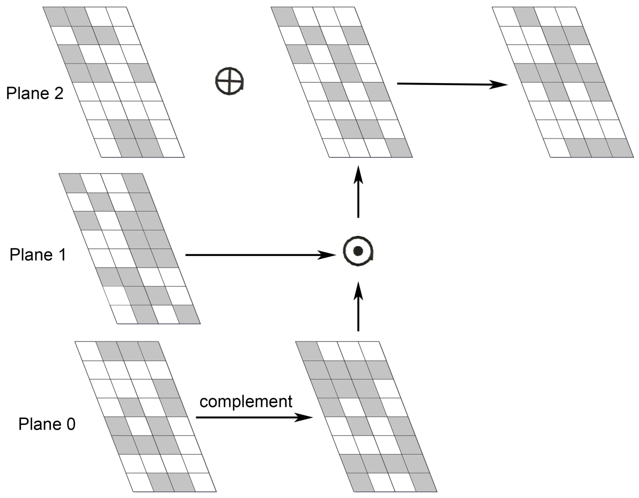

Figure 2.5: An example of the χ function operating on a state

The dispersion layer

A fundamental issue with the χ function and the parity plane step in the θ function is that the interaction between the bits occurs only within the columns of the state. To disperse the bits of the columns among the remaining bits of the state, ρ_{west}, and ρ_{east} were designed as step mappings after the θ function and the χ function respectively. $\rho_w est$ and ρ_{east} together are known as the dispersion layer, and as the name suggests, it helps to disperse the bits all over the state. ρ_{west} is the second step in the round function, it follows the θ function, and is followed by the addition of round constants. The algorithm is as follows:

$$A_1 \leftarrow A_1 \lll (1,0)$$
$$A_2 \leftarrow A_2 \lll (0,11)$$

The ρ_{east} function is the last step in the round function following the χ function. The algorithm is as follows:

$$A_1 \leftarrow A_1 \lll (0,1)$$
$$A_2 \leftarrow A_2 \lll (2,8)$$

Figures 2.6 and 2.7 below shows an example of the ρ_{west} and ρ_{east} operations on a state. The state is separated into three planes, and the first plane remains the same. Bit shifts are performed on the second and third planes according to the shift values, and the state is formed again from the three planes.

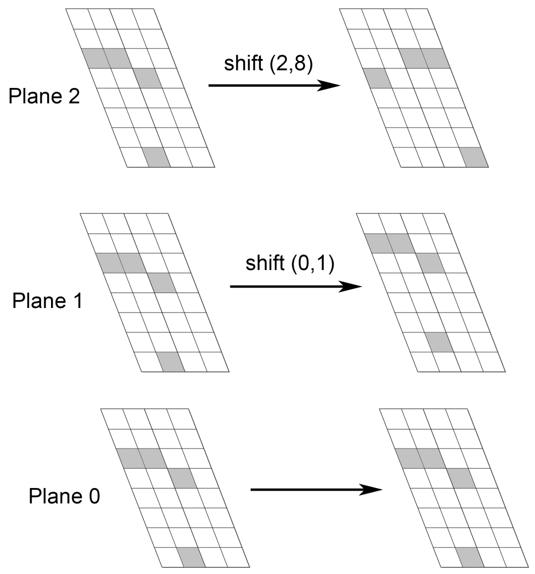

Figure 2.6: Illustration of ρ_{west} on a state

Addition of round constants

The addition of round constants is a critical step in the round function of Xoodoo. It is the third step mapping, after ρ_{west} and before the χ function. The remaining four steps

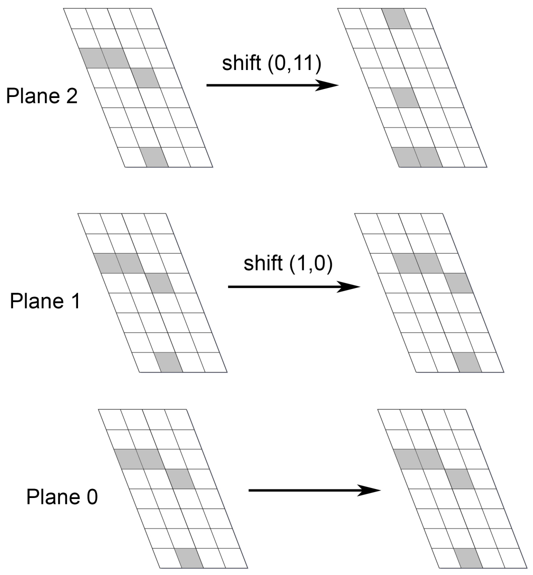

Figure 2.7: Illustration of ρ_{east} on a state

of the round function exhibit a property known as translation invariance, and this is explained in detail in the section below. Translation invariance leads to high amounts of symmetry. Symmetry is another concept explained in the sections below. This symmetry is removed by the addition of round constants. Symmetry also has another weakness: it allows slide attacks. This shortcoming also is removed by the addition of round constants. We can say that the round constants were chosen such that the shift-invariance of the round function is destroyed. The round constants for 12 rounds of Xoodoo are compiled into the Table 2.1 below.

Table 2.1: Round constants for the 12 rounds of Xoodoo

Round i	**Constant** C_i
-11	0x00000058
-10	0x00000038
-9	0x000003C0
-8	0x000000D0
-7	0x00000120
-6	0x00000014
-5	0x00000060
-4	0x0000002C
-3	0x00000380
-2	0x000000F0
-1	0x000001A0
0	0x00000012

During the course of this research, we only work with the naive Xoodoo round function so that we can make use of the symmetry. We define a naive round function as follows.

Definition 1. *The naive round function is the round function of a cipher or a permutation with the round constants stripped off.*

Chapter 3

Related Work

In this chapter, we analyze a few published papers and reports by summarizing their works. These papers or reports are either directly or indirectly related to the main topic of this thesis. The following papers have been considered and elaborated on:

- The Keccak Reference by Bertoni et al. [3]
- Gimli: a cross-platform permutation by Bernstein et al. [2]
- Farfalle: parallel permutation-based cryptography by Bertoni et al. [4]
- A Cryptanalysis of PRINT CIPHER: The Invariant Subspace Attack by Leander et al. [14]
- Is DES a Pure Cipher? by Kaliski et al. [10]
- Column Parity Mixers by Stoffelen and Daemen [25]

At the end of each section, we also discuss how each of these scientific reports relates to our research on Xoodoo.

3.1 Keccak reference

Keccak is a cryptographic primitive designed by Bertoni, Daemen, Peeters, and Assche. It consists of a family of functions based on the sponge construction, which is a mode of operation that operates on a fixed-length permutation. But although the permutation is of a fixed length with the help of padding, the input and output can be of variable lengths. The paper introduced a total set of seven permutations, and similar to Xoodoo, it operates on a finite state by iteratively applying the inner permutation to it for a given number of times.

3.1.1 The construction

This sub-section summarises the Keccak-f input state, the round function, and the construction. Keccak-f[b] is the naming convention to indicate the seven different permutations of Keccak, where $b = 25 * 2^l$, and l can take the values $0, 1, 2, 3, 4, 5$, and 6. b is known as the width of the permutation, and the bits of the state are numbered from 0 to $b-1$. These bits are inserted into what is called a state, which is made up of rows of 5 bits, columns of 5 bits, and lanes that are varying in size. A lane consists of w number of bits, where $w = 2^l$. For a state S, a bit in a given position is represented by the following naming convention: S[x][y][w]. The construction of Keccak-f consists of a sequence of five steps that are iterated for a given number of rounds. The round function can be briefly written as

$$R = \iota \circ \chi \circ \pi \circ \rho \circ \theta$$

where each of the symbols represents a step mapping. Each of these steps manipulates the bits of the state mostly using basic or Boolean algebraic operations. The number of rounds that this round function is permuted is calculated based on the width of the permutation according to this equation: $n_r = 12 + 2l$.

3.1.2 Design rationale

Keccak was motivated by various design choices. Some of the more important ones are listed below.

- The sponge function in Keccak is a simplistic and flexible model that can also provide a variable-length output. This means that the same function can be used for different output lengths.
- The permutations in Keccak-f also have many advantages. Keccak was designed for use in constrained environments, and the concept of symmetry allows the code in software to be very compact.
- It also allows for parallelism, thus making Keccak suitable for fast hardware implementations and CPU pipelining.
- Each of the steps in the round function is of algebraic degree two, and this allows the usage of powerful protection in both software and hardware against differential power analysis and attacks.

3.1.3 Xoodoo and Keccak

The design of Xoodoo was inspired by the design of Keccak, and hence both of them use a "state" as an input, and a state can be broken down to planes, rows, columns, lanes, sheets, and bits. Similar to Xoodoo, the step mappings of Keccak-f's round function are translation invariant in the z-axis direction. This is of course when the addition of round constant step is ignored. Thus, the property of symmetry also follows. The symmetric states are named as having a 'Matryoshka structure'. Such a structure is also observed in Xoodoo, as the symmetry states form a partition of the state space within each input size. This Matryoshka structure allows the analysis of smaller versions of Keccak to be applicable for larger versions. Thus, we implement the smaller versions of Xoodoo to save computational power and time, and the results can still be generalized to the larger versions.

As the design and properties of Keccak and Xoodoo are pretty similar, the Keccak Reference report proves to be one of the most important references for this thesis.

3.2 Gimli: a cross-platform permutation

Gimli is a permutation function that can be used to build high-security block ciphers, stream ciphers, stream ciphers, message authentication codes, hash functions, and many more. Gimli was designed to cover specific design requirements, and these design rationales and the construction of the permutation are covered in the following sections.

3.2.1 The construction

This section talks about the design of a state in Gimli and the construction of the round function. A state in Gimli is made up of 384 bits and is represented as a parallelepiped. It is made up of a 3 x 4 matrix of 32-bit words. Gimli applies this state to a sequence of rounds. Each of these rounds is made up of a non-linear layer, a linear layer and the additions of round constants. Each instance of Gimli runs for 24 rounds.

3.2.2 Design rationale

Many design rationales distinguish Gimli from other permutations. Some of them are listed below.

- Gimli is designed to achieve cross-platform performance.
- It can be used to build ciphers that provide high security and high performance.
- Gimli supports energy-efficient hardware.
- The design of Gimli is such that it can be used in hardware that protects against side-channel attacks.
- It can also be used in micro-controllers, FPGAs, ASICs, and many more.

3.2.3 Gimli and Xoodoo

As we can notice, the construction of Gimli, Keccak, and Xoodoo are all similar and related. Xoodoo takes on the design of Keccak and the input size of Gimli. The property of Gimli that allows the state size of 384 bits and the round functions to blend well into low-end 32-bit processors, vectorization and dedicated hardware is an attractive feature that was adapted into Xoodoo. Thus the Xoodoo permutation has the same width and a Gimli-shaped state also with similar objectives as Gimli, but different (better) propagational properties than Gimli.

3.3 Farfalle: parallel permutation-based cryptography

As the name reads, Farfalle is a permutation-based construction that is used for building pseudorandom functions. This paper was written by Bertoni et al. [4] in 2017. Kravette, an instant of Farfalle based on Keccak-p with 6 rounds is also introduced in this paper. The following sections expand briefly on the construction and modes of use of Farfalle, its design rationales, and its importance to Xoodoo.

3.3.1 The construction

The pseudorandom function is built from a *b*-bit permutation (or a family of permutations) with varying number of rounds. The PRF takes as input a sequence of arbitrary-length data strings and similarly outputs an arbitrary-length output. It also takes a key as an input. The construction of Farfalle is such that there exists a mask derivation, a compression layer, and an expansion layer, and a permutation runs in parallel on all three applications. Each of the input blocks is blinded with a rolling *b*-bit input mask. It operates on strings of bits and makes use of four cryptographic permutations that operate on these input bits - one for the initial mask, one in the compression layer, one between the compression layer and the expansion layer, and the last one in the expansion layer. There are also two "rolling functions", which is again basically a permutation - one that is used for generating masks for the input blocks of the compression layer, and another for the internal state in the expansion layer.

3.3.2 Modes of use

There are three possible modes of use on top of Farfalle. The first is a confidentiality-providing, session-supporting authenticated encryption scheme. The second is a synthetic initial-value authenticated encryption scheme which makes use of a unique combination of a key and a nonce. The third is a wide block cipher which can use a custom block size.

3.3.3 Design rationale

Farfalle is designed to be a very versatile permutation. An instance of Farfalle is also considered efficient as the construction imposes fewer requirements than typical block cipher constructions on the under-laying primitive. The most important property of Farfalle is that it allows three modes, and a high level of parallelism. It supports the parallel implementation of both the compression layer and the expansion layer.

3.3.4 Xoodoo and Farfalle

The paper on Xoodoo introduces Xoofff, which is a primitive that makes use of the Farfalle construction using the Xoodoo as its building blocks. This contributes to making Xoofff an efficient cryptographic function that can be used for encryption, authentication, and authenticated encryption. Thus, the Xoodoo permutation which is a mix of Keccak and Gimli is used in the Farfalle construction to achieve the best possible combination of properties.

3.4 A Cryptanalysis of PRINT CIPHER: The Invariant Subspace Attack

PRINTcipher[12] was introduced by Knudsen, Leander et al and is a block cipher introduced to be used in integrated circuit printing. It works as a light-weight encryption solution for printable circuits, and allows the processing of blocks of two different sizes, namely a 48-bit input block and a 96-bit input block. On analyzing this cipher, Leander et al. [14] discovered and introduced a new shortcoming in the cipher called the invariant subspace attack.

3.4.1 The attack

The cipher is a class of two SP networks, and makes use of an XOR key and a permutation key. Let us consider that there exist cosets of some subspace $\mathbf{F}_2^n$ to which the input belongs. The attack takes advantage of the fact that the round function of the PRINT cipher maps some cosets of the subspaces of $\mathbf{F}_2^n$ to itself. The paper thus went on to show that there exist specific keys that are made up of several fixed bits in specific positions and the remaining being arbitrary values. Considering these keys to be contained in a subspace of the overall sample space, the authors of the paper noticed that the round function maps the subspace on to itself for specific such keys. The same attack was applied to both variants of the PRINT cipher, that is, the 48-bit block and the 96-bit block. An example of one such XOR key for the 48 block variant is given below as:

$$01{*}{*}11{*}{*}{*}{*}{*}{*}01{*}{*}11{*}{*}{*}{*}{*}{*}01{*}{*}11{*}{*}{*}{*}{*}{*}01{*}{*}11{*}{*}{*}{*}{*}{*}$$

The corresponding permutation key for the given XOR key is given as the following:

$${*}11{*}{*}{*}{*}1001{*}{*}{*}{*}11{*}0{*}{*}{*}{*}{*}011{*}{*}{*}{*}$$

For both these keys, $*$ is set to an arbitrary value in $\mathbf{F}_2$, and therefore could either be a 0-bit or a 1-bit.

3.4.2 Results of the attack

To summarise, if the round function maps an input to an output belonging to the same coset, then this shows a weakness in the cipher, as the possible sample space for the output of the cipher, is reduced drastically, thus making it vulnerable and easy to break. The authors of this paper pointed out an interesting property of the cipher, that a single bit difference in the input leads to a difference in the corresponding bit in the output permutation with a probability of $2/8$. Two other noted properties were that there exist s-boxes in the PRINT cipher where the output bits of the s-boxes map onto some input bits of the same s-boxes in the next round and that there exist s-boxes where the round counter called RC_i is not involved entirely. During this process, they identified many weak keys from the keyspace of the PRINTcipher.

3.4.3 Xoodoo and the invariant subspace attack

The invariant subspace attack in the case of Xoodoo would exploit the fact that the round function of the cipher maps a particular subspace (to which the input belongs) to itself. In this way, it is possible to have a better understanding of the cipher and its attack vectors. If it is possible to map a specific input or an input pattern to a set of outputs, it might be possible to spot weaknesses in the cipher.

3.5 Is DES a Pure Cipher?

This paper by Kaliski, Rivest, and Sherman [10] gathered the results of eight cycle experiments on the Data Encryption Standard in the year 1985. The main research question that this paper aimed at answering was whether DES is in fact, a pure cipher. The authors of the paper concluded with high confidence that DES was not a pure cipher. Apart from this, the paper also speaks about observations revealing an unpublished additional weakness in the weak keys of DES.

3.5.1 What is a pure cipher?

If DES was a pure cipher, then Tuchman's multiple encryption scheme would be equivalent to a single encryption. Suppose that three keys i, j, and k are chosen independently. Tuchman's scheme encrypts an input message x by computing the following.

$$T_iT_j^{-1}T_k(x)$$

In this equation, T_n is the Tuchman encryption, that was introduced during a talk presented by Tuchman at the National Computer Conference in June 1978[27].

3.5.2 Cycling experiments

The following tests were conducted as a part of the cycle experiments on the DES cipher.

- The first test that was performed was called a purity test where any transformation $T_0 \subset \tau$ was selected out of the set of transformations, and the cycling closure test was applied to it. This calculates $T_0^{-1}\tau$.
- The next test was called an orbit test, where for any message x_0, they compute x_i such that $x_i = T_k^i(x_0)$ where $i = 1, 2, ...$ for either a specified number of steps, or until a cycle is detected.
- The third test that was performed was called a small subgroup test. Here the cycling closure test is applied to the set of translations T_i, T_j on an input message x_0.
- Extended message space closure test performed the cycling closure text with an extended message space using a small integer $l > 1$.
- The last test implemented was called a reduced message space test where the message space is reduced in size.

With many combinations of these five tests, it was found that the cycle of a message under the composition of two weak keys of DES resulted in a small cycle of length less than 2^{33}. The probability of this occurring was extremely low and was calculated to be 2^{-37}. The orbit test that we saw in the previous sub-section is most similar to the implementation that is followed in this academic paper. In the later chapters, we will notice that in Xoodoo, we try to detect cycles similar to what is called the orbit test in this paper.

3.6 Column Parity Mixers

Column parity mixers [25] (CPMs) are a generalization of the θ function or the mixing layer that we saw in Keccak, Gimli, and in Xoodoo. These CPMs operate on two-dimensional arrays, and as the name suggests, the parity of the columns plays an important role in the same. They make use of linear transformations operating on the column parity of a matrix. This is called the parity-folding transformation. This parity-folding matrix is then used to define the θ-effect of the matrix that was taken into consideration. The column parity mixer is calculated by computing the expanded θ-effect of the matrix and adding that to itself.

3.6.1 Design rationale

A few design specifications are as follows.

- CPMs were designed and can be used as an alternative to Maximum distance separable matrices (or MDS matrices).
- CPMs can prove to be lightweight mixing layers.
- They work well with bit-sliced implementations, 4-bit ciphers, and permutations.

3.6.2 CPMs in Xoodoo

We can see from the previous chapter that the mixing layer in Xoodoo is constructed similarly. The first step of the mixing layer or the θ function is the parity-folding transformation as mention above, although its calculation is different from what we saw in the CPM paper: the parity plane is the resulting XOR of the three planes. The θ-effect plane in the case of Xoodoo is a shifted version of the parity-folding plane, and it is added to a sheet x. The resulting column parity mixer depends directly on the sheet $x-1$, just as it was explained in the sub-section about CPMs.

Chapter 4

Symmetry and its properties

In this chapter, we delve into the concept of symmetry. We first introduce translation invariance, which is an important property of symmetry, and we describe it in this chapter with examples and definitions. Symmetry is then explained in order to understand the Xoodoo states and how they are formed. We introduce lattices and sub-lattices, and speak about the many symmetry classes of Xoodoo. We also explain how the shift positions are calculated for different symmetry classes, and move on to define super-states. A Xoodoo state also exhibits symmetry within the state, and the three types of symmetry are presented. In the next chapter, we will move on to the methodology used to perform our research.

4.1 Translation invariance

The four steps of Xoodoo - namely the mixing layer, the non-linear layer function, rho west, and rho east all operate on the bits of Xoodoo irrespective of their position on the plane. This means that a state can be in its original form or it can be shifted by say, x positions in a given direction, and the naive Xoodoo round function will have the same effect on the bits of the state irrespective of the shifts, such that the output of the shifted state is simply a shifted version of the original output.

A translation moves each point on a space by a specific distance in a given direction. It adds a constant vector to each point, thus shifting the origin of the coordinate system itself. For a given vector v, a translation T over a constant c can be denoted as the following:

$$T_v : (c) = c + v$$

Suppose that τ is a mapping that translates bits in a Xoodoo state A. If the mapping causes a translation of the bits of the state in the three directions by positions x, y, and z, then we can represent this mapping over the state A as the following:

$$A \lll (x, y, z)$$

Next, to understand translation invariance, we consider τ to be a mapping that translates the Xoodoo state by a single bit in the horizontal direction, such that $B = \tau(A)$, where A and B are states of Xoodoo. Let w be the width of the state A. The translation over the state A would be as follows:

$$\begin{aligned} B[x][y][z] &= A[x+1][y][z], \text{ if } 0 < x < w \\ B[x][y][z] &= A[0][y][z], \text{ if } x = w \end{aligned} \tag{4.1}$$

We can generalise this for a translation over t bits as:

$$B[x][y][z] = A[(x+t)\text{mod } w][y][z]$$

Where l is the length of the state, we can generalise a translation over t bits in all three directions as:

$$B[x][y][z] = A[(x+t)\text{mod } w][(y+t)\text{mod } 3][(z+t)\text{mod } l]$$

Now, we can define translational invariance.

Definition 2. *A mapping α is translation invariant in three directions x, y, and z over the translation $\tau_{(x,y,z)}$ if*

$$\tau_{(x,y,z)} \circ \alpha = \alpha \circ \tau_{(x,y,z)} \tag{4.2}$$

Both the θ and χ functions are found to be translation invariant in all three directions. It is also noticeable that the rho west and rho east functions give rise to no column-wise interaction, and this makes them translation invariant in all directions where y=0. The four step mappings discussed above are not just translation invariant, but also invertible.

4.2 Symmetry

Translation invariance is a property of another concept known as symmetry. To understand symmetry better, we first introduce a few other concepts. Let us take into consideration an input state of Xoodoo. The naive Xoodoo round function can be applied to three-dimensional states of any width and length. This can start with the smallest possible values up till infinity. Any state with finite width and finite length can be seen as a state that is periodic in the x and z directions. A periodic function is one that repeats its values in regular intervals or periods. We can see the Xoodoo state as an infinite state with a period of 4 in the x-direction, 32 in the z-direction and 3 in the y-direction.

Consider a vector space V. A set of vectors in V can be called a basis if every element in V can be written as a linear combination of the elements of the basis vectors, where the basis vectors are the set of all elements of the basis [26]. For any given basis, the set of all linear combinations with integral coefficients of the basis vectors forms a subgroup s. In other words, a lattice is a set of vectors that can be generated by a basis. Although there can be no non-trivial linear combination of basis vectors that is zero.

The basis vectors for the Xoodoo lattice are:

$$\langle (4,0,0),(0,3,0),(0,0,32) \rangle \tag{4.3}$$

The Xoodoo state is periodic in three dimensions fulfilling the property that if it is shifted over any vector in the lattice, then it is mapped on to itself. Thus, due to shift-invariance or translational invariance, the naive Xoodoo round function maps Xoodoo states to Xoodoo states. The Xoodoo states are invariant over a shift in the Xoodoo lattice, and this property can be called as symmetry, or also periodicity. Although in the algorithm of Xoodoo the shifts are cyclic, here we imagine an infinite state periodic in three directions, thus the shifts are discrete in the infinite three-dimensional space, but not cyclic.

If we consider the effects of shifts in the finite representation of a Xoodoo state, then we can see that the shifts become cyclic. Here, a state is represented as a finite three-dimensional rectangular array. If we visualize the shifts as arrows, then the end point of the shifts must end in the array. Calculation of the shifts is a reduction modulo the lattice, so that the shift arrows end up in the three-dimensional array.

A sub-lattice is a group of translation operations that have a subgroup relation to another lattice. More formally, it is a subset of an existing lattice L, such that the meet and join operations are the same as the ones in L. Meet is the greatest lower bound or the infimum, and join is the lowest upper bound or the supremum. For any elements in a subset of the lattice, if the meet and join operations are in the subset itself, then the subset can be called a sub-lattice[19]. Consider a vector space V', where the Xoodoo lattice is a sub-lattice of V'. Here, again, if a state is invariant over a shift, meaning it is translation invariant, then it is a Xoodoo state due to periodicity that is caused by our perception of the three-dimensional infinite state. Let us now define a sub-lattice.

Definition 3. *A sub-lattice of a lattice is a non-empty subset where for any two elements in the subset, the meet and join operations are the same as the lattice's meet and join operations.*

The Xoodoo lattice $\langle(4,0,0),(0,3,0),(0,0,32)\rangle$ is both a lattice in itself, and can be the sub-lattice of another lattice. Let us consider a subset of elements in the state space S. If a mapping over S is translation invariant, then the elements of S form a symmetry class. In other words, a shift-invariant mapping maps elements in a symmetry class to elements in the same symmetry class. Each symmetry class forms a partition of the state space of Xoodoo. In the case that the Xoodoo lattice is indeed a sub-lattice, we can notice that there is a layer of symmetry within this symmetry class. Thus, when the Xoodoo lattice is a sub-lattice, more symmetry is observed.

According to [7], a state belongs to a specific symmetry class S_V where V is the Xoodoo lattice if two conditions are satisfied.

- The first condition is to satisfy translation invariance along V.
- the second condition is that there exists no other lattice V' such that $V \subset V'$ and the state is invariant with respect to V'. Thus the state must belong to a unique symmetry class.

The symmetry classes form a partition of the state space, and are specified based on their lattices. Let us consider the sets of states that are periodic according to some lattice as T_V. Then, T_V is the union of S_V and $S_{V'}$ corresponding to all lattices V' for which V is a sub-lattice of V'. Hence, T_V is the state space of Xoodoo with V being the Xoodoo lattice.

For the remainder of this section, we omit the y-coordinate as it is a constant, and concentrate on the other two. If the first element is of the form $(0, 2^e)$ where e can take values 0, 1, 2, 3, 4, and 5, then the second element can be the following:

- For all values of e, there exist the following pairs of basis vectors where the second element can be $(4,0)$, $(2,0)$, and $(1,0)$.
- For all values of e that are greater than 0, there exist two pairs of basis vectors, where the second element can take the form $(2, 2^{e-1})$, and $(1, 2^{e-1})$.
- For all values of e that are greater than 1, there exist two more pairs of basis vectors, where the second element can be of the form $(1, 2^{e-1})$, and $(1, 3.2^{e-2})$.

This gives a total of 36 lattices including the original basis vector of (4, 0) and (0, 32). For the case where $e = 0$ and the basis elements are $\langle(1,0),(0,1)\rangle$, there are two sub-symmetry classes. This can be explained by the fact that the two states that contain either only 1 or only 0 bits are also shift-invariant along the y-coordinate, so these two states form a symmetry class of its own. This amounts to a total of 37 classes. It is also clear that the symmetry classes do not always have the same state dimensions. This can be justified by the use of different basis vectors for each case, therefore leading to different periods on the infinite state.

Therefore we can say that due to shift-invariance, applying a step function from the naive Xoodoo round function on a state in a symmetry class will result in a state within the same symmetry class, consequently concluding that these symmetry classes are invariably invariant subsets. This is explained in detail with an example in Chapter 5.5.

4.3 Symmetry classes and their shifts

We understood from the previous chapter that there are a total of 37 symmetry classes. The naive Xoodoo round function operates on each of these symmetry classes. Three out of five step mappings involve the shifting of bit positions within the planes of a state. These bit shifts vary from 0 position shifts to up to 14 position shifts. It would be interesting to note that some of the input states in a symmetry class might have dimensions less than the number of shifts that are required. As the basis vectors are different, the shifts that we observe in the round function are also different. The symmetry classes with smaller dimensions are visualized as an infinite state periodic where the state is constantly repeated, and the shifts must be reduced to the three-dimensional array module the lattice. For each symmetry class, the shifts are calculated using the basis vectors. The basis vectors are multiplied by linear integral coefficients and then is added to the initial shift values. To understand how this is calculated, we assume the example of a symmetry class with basis vectors $\langle(1,24),(0,32)\rangle$. In the second sub-step of the ρ_{east} mapping, the state is shifted by $(2,8)$ positions. But, a shift of 2 positions in the x-direction is not possible in the chosen symmetry class. The new shifts are hence calculated as follows:

- Find a linear coefficient for one of the basis vectors such that when added with the shift vectors (2, 8), the resulting x-coordinate shift is mathematically possible. We calculate this coefficient to be -2.

$$\begin{aligned}&(2,8)-2(1,24)\\&=(0,-40)\end{aligned}$$

- Find a linear coefficient for the other basis vector such that when it is added with the resulting vector from the previous step, a shift within the boundaries of the possible width and length are found. We calculate this coefficient to be 1.

$$\begin{aligned}&(0,-40)+1(0,32)\\&=(0,24)\end{aligned}$$

- We thus find (0, 24) to be the correct shift for the chosen symmetry class instead of (2, 8).

This method need not be followed for all symmetry classes. If the basis vectors of the symmetry class are of the form $\langle (x,0),(0,z)\rangle$, then the updated shift would be the original shift modulo length or width, as per requirement. Therefore for a symmetry class with basis vectors $\langle (4,0),(0,8)\rangle$, the (2, 8) shift indeed becomes (2, 0). Table 4.1 displays shift values for 20 symmetry classes of Xoodoo.

4.4 Xoodoo super-states

An equivalence relation that is equality modulo horizontal shift defines a partition of elements known as the super-states. The super-state set contains a set of states that are equal modulo horizontal shifts. For an input of 384 bits, there are 128 bits in each plane, thus there would be 128 equivalent shifted states. To generalize, for any input state that has n bits in each plane, there would be n different shifted states and are equivalent to each other, where $n = width * length$. If $a_0, a_1, ..a_n$ represent a set of super-states, then the equivalence relation R for the same is given below as:

$$(a_0, a_1, ..a_n) \in R \forall a_i \in V, 0 < i < n \tag{4.4}$$

V is the lattice, and R is symmetric.

Definition 4. *A super-state is set of states that are equivalent modulo a shift, and they form a partitioning of a symmetry class. For any two states A and A' in a super-state set, there exists a shift vector $(x,y,z) \in V$ such that*

$$A = A' \lll (x,y,z) \tag{4.5}$$

Table 4.1: Symmetry classes of Xoodoo

Size	**Width and length**	**Base vectors**	**θ one**	**θ two**	**ρ_{west} one**	**ρ_{west} two**	**ρ_{east} one**	**ρ_{east} two**
3	1 x 1	(1,0) (0,1)	(0,0)	(0,0)	(0,0)	(0,0)	(0,0)	(0,0)
6	1 x 2	(1,1) (0,2)	(0,0)	(0,1)	(0,0)	(0,1)	(0,1)	(0,0)
	1 x 2	(1,0) (0,2)	(0,1)	(0,0)	(0,0)	(0,1)	(0,1)	(0,0)
	2 x 1	(2,0) (0,1)	(1,1)	(1,0)	(1,0)	(0,0)	(0,0)	(0,0)
12	1 x 4	(1,3) (0,4)	(0,2)	(0,3)	(0,1)	(0,3)	(0,1)	(0,2)
	1 x 4	(1,1) (0,4)	(0,0)	(0,1)	(0,3)	(0,3)	(0,1)	(0,2)
	1 x 4	(1,2) (0,4)	(0,2)	(0,0)	(0,2)	(0,3)	(0,1)	(0,4)
	2 x 2	(2,1) (0,2)	(1,1)	(1,0)	(1,0)	(0,1)	(0,1)	(0,1)
	1 x 4	(1,0) (0,4)	(0,1)	(0,2)	(0,0)	(0,3)	(0,1)	(0,0)
	2 x 2	(2,0) (0,2)	(1,1)	(1,0)	(1,0)	(0,1)	(0,1)	(0,0)
	4 x 1	(4,0) (0,1)	(1,1)	(1,0)	(1,0)	(0,0)	(0,0)	(2,0)
24	4 x 2	(4,0) (0,2)	(1,1)	(1,0)	(1,0)	(0,1)	(0,1)	(2,0)
	2 x 4	(2,0) (0,4)	(1,1)	(1,2)	(1,0)	(0,3)	(0,1)	(0,0)
	1 x 8	(1,0) (0,8)	(0,5)	(0,6)	(0,0)	(0,3)	(0,1)	(0,0)
	2 x 4	(2,2) (0,4)	(1,1)	(1,2)	(1,0)	(0,3)	(0,2)	(0,1)
	1 x 8	(1,4) (0,8)	(0,1)	(0,2)	(0,4)	(0,3)	(0,1)	(0,0)
	1 x 8	(1,6) (0,8)	(0,7)	(0,0)	(0,2)	(0,3)	(0,1)	(0,4)
	1 x 8	(1,2) (0,8)	(0,3)	(0,4)	(0,6)	(0,3)	(0,1)	(0,4)
384	4 x 32	(4,0) (0,32)	(1,5)	(1,14)	(1,0)	(0,11)	(0,1)	(2,8)

For a state where $width > 1$ and $length > 1$, the shifts are found to be cyclic in two dimensions. We can also observe the shifts to be a modulo horizontal shift and a modulo vertical shift. Each shifted version of the state would be a modulo horizontal or vertical shift, or a combination of the two, which is also called a 'diagonal shift'. Figure 4.1 represents four states that are equivalent to each other due to 1 modulo vertical shift.

4.5 Symmetry in a state

When two halves of a state are the same, we say that it is symmetric. A Xoodoo state can exhibit three different types of symmetry, all of which involve the value of the bits that the state is made up of. The three types of symmetry are lengthwise symmetry, transversal symmetry, and skew symmetry. It is important to note that for all cases, a state is said to be symmetric only if all the three planes have the same symmetry.

4.5.1 Lengthwise symmetry

Let us suppose that a plane is divided into two halves by a horizontal line. If the line divides the plane into two halves where the corresponding bits in each are the same, then this plane is lengthwise symmetric. If this same symmetry holds for all three

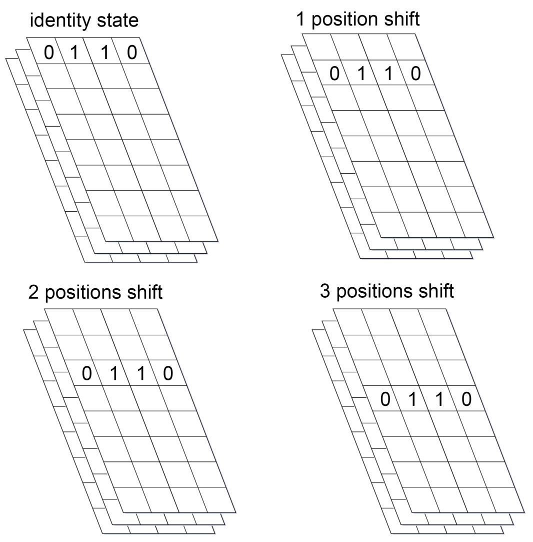

Figure 4.1: Four states in a super-state set.

planes of a state, then we can call that the state is lengthwise symmetric. An example of an 8-bit plane displaying lengthwise symmetry is shown in Figure 4.2 below. Note that each of the letters a, b, c, and d can have bit values of either 0 or 1.

$$\begin{array}{cccc} a & b & c & d \\ \hline a & b & c & d \end{array}$$

Figure 4.2: Lengthwise symmetry in an 8-bit plane.

4.5.2 Transversal symmetry

Transversal symmetry is similar to horizontal symmetry, except that the state is divided into two halves by a vertical separator line. If these two halves are exactly equal in a bit-wise manner, then we can say that the state exhibits transversal symmetry. An example of transversal symmetry is illustrated in the figure below. The vertical line in between is the separator that divides the state into two equal halves.

$$\begin{array}{cc|cc} a & b & a & b \\ c & d & c & d \end{array}$$

Figure 4.3: Transversal symmetry in an 8-bit plane.

4.5.3 Skew symmetry

Skew symmetry is a bit different from lengthwise and transversal symmetry, it is rather a mix of the two. If the state is divided equally by one horizontal line and one vertical line, then, as expected the state would be divided into four parts. If the two sets of opposing halves are equal, then we can say that the state holds skew symmetry. The figure shows the basic format of skew symmetry in an 8-bit plane.

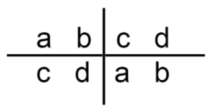

Figure 4.4: Skew symmetry in an 8-bit plane.

Chapter 5

Methodology

The purpose of this chapter is to understand the methodology that is followed to achieve results. The first section explains the idea of cycles in Xoodoo, and describes how cycles are formed. The next section strives to provide an in-depth understanding of the implementation of the naive Xoodoo round function with the help of pseudo-codes. Some factors affect the implementation, and some factors simplify it. To begin with, we present a convention to simplify how a state is displayed. We then see how to avoid a state from being processed by the naive Xoodoo round function for a second time. The later sections explain how the super-states are shifted and how to effectively find and understand the number of super-cycles formed. The last section uses an example to explain why the symmetric states can be ignored during implementation. Overall, we learn about all the factors that go into implementing the naive Xoodoo round function. The results that are derived from this implementation have been summarized in the Chapter 6.

5.1 Xoodoo cycles

The first step towards the implementation of our experiment is a strong understanding of how cycles are formed. The naive Xoodoo round function of the permutation has four step mappings. The bits of an input are simply mathematically manipulated in a bit-wise manner to derive an output of the same length. To successfully find a cycle, a chosen input is permuted by naive Xoodoo round function. Let us consider a state A, and the naive Xoodoo round function to be f. We start with A and keep iterating the function on the input state to form $f(A)$, $f(f(A))$, $f(f(f(A)))$, and so on. This iteration continues, and the loop terminates only when we reach state A again. With this, we define a cycle.

Definition 5. *A cycle is a subset of a permutation that maps the elements of the subset to each other in a cyclic manner.*

Every state that is calculated as an output of the naive Xoodoo round function to find a cycle contributes to the cycle length. Therefore, a cycle of length 30,000 would indicate that the chosen input required 30,000 iterations of Xoodoo to redirect back to the initial input state. This further means that there exist 29,999 states apart from the initial input state that are also involved in the cycle. All states that comprise of a cycle are unique and are also non-repetitive within the cycle. All these 29,999 states would produce the same cycle length of 30,000, and the order in which the states formed a cycle would also remain the same. We now define cycle length.

Definition 6. *The length of a cyclic subset of a permutation is known as its cycle length.*

Each lattice that has the Xoodoo lattice as a sub-lattice defines a symmetry class, and we implement the naive Xoodoo round function in order to understand the cycle structure and report on all the possible cycles of these symmetry classes. Let us now define cycle structure.

Definition 7. *The cycle structure of a given permutation is the list of the cycle lengths formed and their multiplicities.*

We analyse the cycles and report on the number of occurrences of each cycle length, and also the length of super-cycles. A super-cycle is formally defined later in this chapter in Section 5.4.

5.2 The implementation

We have considered and implemented 19 symmetry classes with varying sizes and dimensions. We have only considered 19 out of all possible symmetry classes due to time and computational power constraints. The remaining symmetry classes require intensive computational power due to its very high state size. Many factors contribute to the implementation of the naive Xoodoo round function and they are discussed in this section.

The four step mappings that constitute the naive Xoodoo round function are implemented using the programming language C++. The implementation is coded while taking into consideration all possible symmetry classes for the given version of Xoodoo. Each step mapping is written into a separate function. A for loop starts at the first possible state, which comprises of all 0 bits and goes on until the maximum possible size, which is the state consisting of all 1 bits. This for loop is defined in the *main* function, which looped from the first possible input until the last. Within this for loop, a $do..while$ loop is implemented to find cycles. Within this loop, each of these inputs is processed by the naive Xoodoo round function repetitively until a cycle is found. The cycle length is then noted, and the total number of cycles and states that have been covered is also noted. The loop terminates when either the input state is reached, or a shifted version of the input state is reached. The entire code can be found here [28] for reference. Before introducing a few concepts that influence the states that enter the for loop in the algorithm, a basic pseudo-code of the implementation can be found in Algorithm 2.
The following sections delve into other factors that influence the implementation, namely it introduces a convention used to display the states, provides an explanation on what happens to the states that are involved in a cycle, and explains symmetric states and shifts.

Algorithm 2 Pseudo-code for the implementation of the naive Xoodoo round function

```
Require: $n$ = number of possible input states; $A', A_i$ = states of Xoodoo
  for $i$ from 0 to $n$ do
    $A' = A_i$
    $cycleLength = 0$
    do
      $A'$ = naiveXoodooRoundFunction($A'$)
      $cycleLength += 1$
    while $A' \notin$ superStatesSet($A_i$)
  end for=0
```

5.2.1 A convention to display states

Xoodoo is represented as a three dimensional cuboid with bits evenly distributed on it. The implementation performed in this chapter uses the bits of Xoodoo as follows. Let us consider a symmetry class of Xoodoo where the state is made up of 24 bits. There would be 2^{24} possible values for this state. A state of 24 bits can also be represented as an integer by defining an order of the bits of the state - namely by concatenating all the bits of planes, starting from the plane that is indexed as $y = 0$, and calculating the integral representation of the binary digits. Let us assume that three planes of a state have bits 00000001, 10010100, and 10001110 written into them. This can be interpreted as the integer 103566. To simplify this representation of Xoodoo, we use a convention. Here, the nibble is viewed from a top-down perspective, and for a given position on the planes, the three bits are concatenated and interpreted as an octal representation. An example of the same can be seen in Figure 5.1 below. The computer still stores the state in binary bits, but for the remaining of this thesis, a state is represented as a plane in base-8. This plane can be expanded to three corresponding planes by converting the octal representation to binary, to achieve the initial representation of the state again.

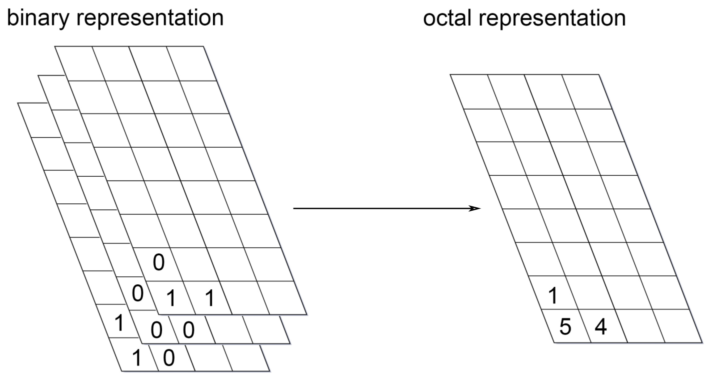

Figure 5.1: Change in representation from a binary state to an octal plane.

5.2.2 Parsed states

The purpose of implementing the naive Xoodoo round function is to find all possible cycle lengths and to find the total number of cycles that would consume all the states of Xoodoo. If a given state takes n iterations to finally reach the same input again, then it can be known for certain that the states that were involved in this cycle will not appear again in any other cycle. This is because all the states that formed the cycle would have the same cycle length. To avoid encountering the same states again, we create a one-dimensional array called $MARKED$ with length equal to the total number of possible inputs. The array is indexed from 0 to $i-1$, where i is the total number of input states. It would represent the state's value in its integral interpretation. The value in the array would be a binary number that stores the value 0 initially. Hence, the entire array is initialized with the value 0. Once a state is sent through the naive Xoodoo round function, this value is changed to 1. As we are also dealing with super-states and not just states, we try to find all shifted versions of a state and mark them in the array as well. Therefore, the original state, along with all of its shifted versions are marked in the array. Hence, before entering into an iteration of the loop, there would be a check that ensures whether the input state has already entered the loop. We have given the pseudo-code for a better understanding of how this array is implemented in Algorithm 3.

Algorithm 3 Pseudo-code for the MARKED array in the naive Xoodoo round function implementation

```
Require: $n$ = number of possible input states; $A_i, A'$ = states of Xoodoo
  for $i$ from 0 to $n$ do
    $MARKED[j] == 0$
  end for
  for $i$ from 0 to $n$ do
    if $MARKED[j] == 1$ then
      continue
    else
      $A'$ = naiveXoodooRoundFunction($A_i$)
      do
        $A'$ = naiveXoodooRoundFunction($A'$)
        $MARKED[j] = 1, \forall\ j$ such that $A_j \in$ superStatesSet($A'$)
      while $A' \notin$ superStatesSet($A_i$)
    end if
  end for=0
```

5.2.3 Implementing symmetry

A separate function is written to check for symmetry within each state that we encounter during the implementation. If a state is found to have the property of length-wise symmetry, transversal symmetry, or skew symmetry, or even a combination of more than one type of symmetry, then the state is not in the symmetry class that we are investigating, and does not have to be involved in the implementation. We do not include it in our final list of cycle lengths. Consider a symmetry class that consists of states with an input size of 6 bits. The 6-bit state with width 2 bits and length 1 bit would display transversal symmetry for a few cases. Similarly, if the width is 1 bit and the length is 2 bits, the state would exhibit lengthwise symmetry in certain cases. These states that exhibit symmetry can be broken down into two columns of 3-bit states. As we would have already covered it in our implementation of the 3-bit states, it would be repetitive to include the symmetric states once again. Therefore, this symmetry check aids in making our code more efficient. A short pseudo-code for the same can be seen in Algorithm 4. Note that in the algorithm, lengthwiseSymmetry, transversalSymmetry, skewSymmetry are functions that check if the state that is a parameter to the function exhibits the corresponding symmetry, and returns 1 if it does.

Algorithm 4 Pseudo-code for the symmetry states not being taken into consideration for the naive Xoodoo round function implementation

```
Require: $n$ = number of possible input states; $A_i$ = states of Xoodoo
  for $i$ from 0 to $n$ do
    if (lengthwiseSymmetry($A_i$) || transversalSymmetry($A_i$) || skewSymmetry($A_i$))
    then
      continue
    else
      $A'$ = naiveXoodooRoundFunction($A_i$)
      do
        $A'$ = naiveXoodooRoundFunction($A'$)
      while $A' \notin$ superStatesSet($A_i$)
    end if
  end for=0
```

5.3 Shifts and super-states

For a state of Xoodoo that is made of 3 planes of n bits per plane, there can be n elements in its super-state set. But, it is not possible to find these super-states the same way each time.

During the cycle experiments, these super-states give way to interesting observations. We notice that if in a plane, either $x = 0$ or $z = 0$, then it becomes a one-dimensional plane. In this case, the shifts are cyclic. They can be seen as states that are equivalent modulo shifts in the direction where the coordinate is not equal to zero.

We also notice that for a case where neither $x = 0$ nor $z = 0$, the shifts are cyclic in two directions. This can be explained by considering every directional shift in the plane which leads to translation invariance. In a plane, the step mappings in the naive Xoodoo round function lead to right shifts in the x direction and forward shifts in the z direction. Thus, if a state is kept constant but shifted in all possible directions (two, here) by all possible shift amounts, then we can create a superset of states. If the x

coordinate and z coordinate have m and n bits each, that is., if the size of a plane in the state is $m*n$, then all possible shifted versions of the state can be found by shifting the original state in all combinations of $[0-m],[0-n]$. This would give a total of $m*n$ states in a super-state set.

5.4 Cycles

This section explains how super-cycles are calculated based on the initial and final state of the cycle. First, we define a super-cycle.

Definition 8. *The collective set of cycles covering all states in a super-state set is known as a super-cycle.*

We consider three examples states because there can be three possibilities when it comes to the construction of a state:

- A state where either $width = 1$ or $length = 1$
- A state where $width = length$ and $width, length > 1$
- A state with $width > 1$ and $length > 1$

5.4.1 A rectangular one-dimensional state

Let us consider the symmetry class of Xoodoo with states of length 1 bit and width 8 bits. We refer to Figure 5.2 for this scenario. The shifted versions for this symmetry class are simply modulo horizontal shifts. Let us name the 8 shifted versions with a_0 being the initial state, a_1 being the state shifted by an offset 1, and so on until a_7 which is shifted by an offset of 7. The *do..while* loop is terminated when any of the states from its super-state set is reached If a_0 is the initial state, the following possibilities can occur.

- First, let us consider the case where the last state is the original input state itself, and not any of the shifted states in the super-state set. This would, in turn, mean that the input state a_1 would reach a cycle with a_1 itself as the last state, and so would all the other states in the super-state set. As there are eight states in the super-state set, they would form a total of eight cycles. Each of these eight cycles would cover the same number of states before it reaches a cycle. This is because of symmetry. Every state that is an intermediate state for the cycle of a_0 will also occur in the same position for all other states a_1, a_2, and so on but with the same corresponding shift that the input has from a_0. Thus, if we say that each cycle has a length of n, we can conclude that a total of $8n$ states would have been encountered to cover all cycles for all states within the super-state set.
- Next, let us consider the case when the input state a_0 forms a cycle with the last state as a_4. With an offset of 4, we reach state a_4 from state a_0, and from a_4 with the same offset of 4, we will at some point again reach a_0. Thus, a cycle can be formed similarly with input state as a_1 ending in state a_5 with an offset of 4. To

cover all eight states in the super-state set, a total of four such cycles formed. In the previous point, we established that there are a total of $8n$ cycles. In this case instead, there would be four cycles, each of length $2n$.

- The third case is if the last state in the cycle for an input a_0 is the shifted version a_2. If an initial state ends in a state with an offset of two, this means that if the *do..while* loop is not terminated once it reached a shifted state, then after the state a_2, a_4 and a_6 would also be present in the same cycle. In other words, four shifted states are covered in a single cycle. As there are a total of only eight shifted states, we can conclude that there would be a total of two cycles when the offset is two. Again, as we established the total number of states covered as $8n$, the two cycles, in this case, would be of lengths $4n$ each.
- The last possible case explains the result when the offset is anything but 0, 2, or 4. That is, when the offset is 1, 3, 5 or 7, then this would lead to exactly one cycle of length $8n$. This is because all shifted versions will be covered in that singular cycle due to the odd offset size.

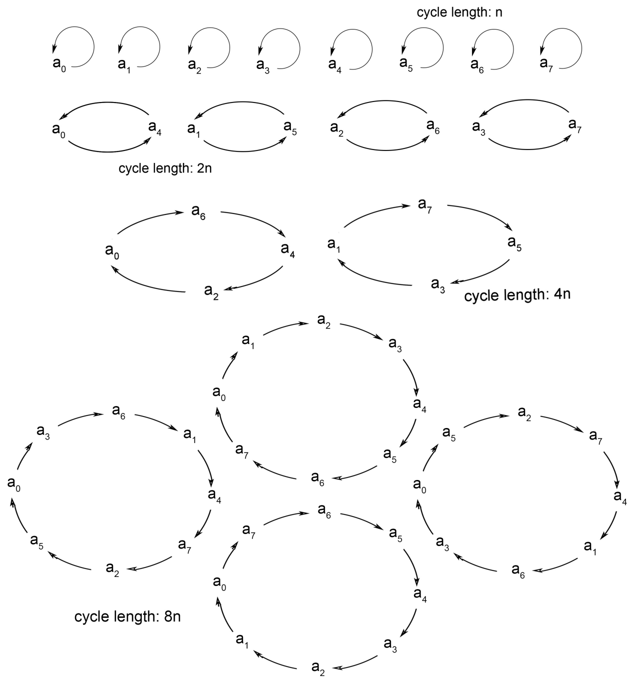

Figure 5.2: Cycles of a one-dimensional 24-bit input

5.4.2 A square state

For this, we take the symmetry class with dimensions $length = 2$ and $width = 2$ into consideration. This is of size 12 bits. There are four bits in each plane, meaning there are a total of four shifted states in a super-state set. Again, if the initial state is called a_0, then the state with a single shift in the horizontal direction is called a_1, the state with a single shift in the vertical direction is a_2, and the state with one vertical and one horizontal shift is a_3. This is much simpler than the previous case, both due to the smaller size and due to the two coordinates being equal. We analyze the cases one by one.

- Consider the case where cycle with the input state a_0 terminates at a_0 itself, meaning no other state within the super-state set is encountered in the cycle. Thus similar to the first case of the previous construction, there would be four cycles of length n each, hence a total of $4n$ states are covered in this process.
- For the other three cases where the offset is either 1, 2, or 3, there would be two cycles. This is because none of these offsets can indeed cause a cycle with more than two states per cycle. This is because when the x and z coordinates have an equal number of bits, the shift is cyclic in both x and z directions. Thus it is different from a cyclic shift that we saw in the previous scenario. These two cycles would be of length $2n$ each, thus forming a total of $4n$ states as seen above. In Figure 5.3, we can see the cycles that can occur when the offset is anything but zero.

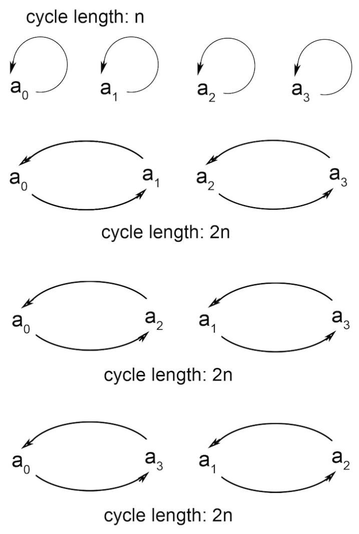

Figure 5.3: Cycles of a square-shaped 12-bit input

5.4.3 A rectangular two-dimensional state

To understand how many cycles are formed in the super-state set of a state with $x, z > 1$, we consider a symmetry class with width 2 bits and a length of 4 bits. This is a 24-bit state with 8 bits per plane, organized in a rectangle with a length of four bits and width two bits. Similar to previous naming conventions, we name the first initial state as a_0, and so on until a_7. These eight shifted states are explained in Chapter 6.4, and can also be viewed in Figure 6.6.

- Just as explained in the previous two scenarios, if the initial state and the final state are the same, with no intermediate states belonging to the super-state set, then there would be a total of eight cycles of length n each, thus covering a total of $8n$ states.

- If the final state is reached after an offset of two, then a_0, a_2, a_4, and a_6 will fall in the same cycle. The remaining four states will form another cycle. Thus there would be a total of two cycles of length $4n$ each. This is also the case for an offset of 6. The order of course, would be different - a_0, a_6, a_4, a_2. The other cycle would contain the remaining four states.
- For an offset of 1, 3, 4, 5, and 7, there would be four cycles of length $2n$ each. This is due to the structure of the plane, and the shift directions. A cycle would not contain more than two states from the super-state set. Thus we would need a total of four cycles to cover eight states. Each of these cycles are visualized in Figure 5.4.

We encounter only one case where the number of cycles is odd, and this is if for an odd offset in a plane which either has the x dimension as 1 or the z dimension as 1. All other cycles occur in pairs, quadruplets, or sets of eights.

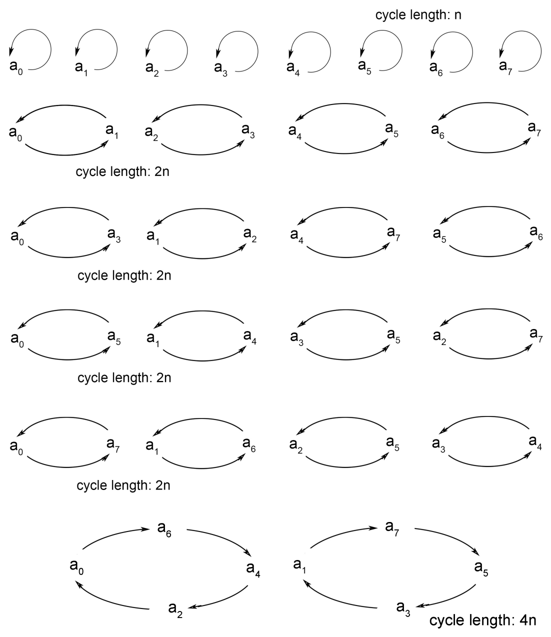

Figure 5.4: Cycles of a two dimensional 24 bit input

5.5 Symmetry within a state explained

Consider a 6-bit state. The input with width and length coordinates 2x1 displays transversal symmetry in some cases. Similarly, the 1x2 input displays lengthwise symmetry in certain cases. It would be interesting to note that if the symmetric states were not removed from the implementation, rather only the symmetric states were executed to find cycles, then the result would be the same as the result of the 3-bit version. There would be 5 cycles, with the same states resulting in the same cycle structures. Similar behavior can be found in the 12-bit scenario, only a bit more complex. It involves some cycles from the 6-bit version, and some also from the 3-bit version. For the 24-bit state, the symmetry states involve cycles from the 3-bit, 6-bit, and 12-bit version, and so on.

To understand how a symmetric 24-bit state behaves, let us consider the example provided in Figure 5.5, it represents a 24-bit state that exhibits all three types of symmetries. We see that the cycle length is 6. This state would have the same cycle length as the 6-bit state that is seen to repeat itself in the 24-bit construction.

This state displays lengthwise symmetry and transversal symmetry. Let us take advantage of this transversal symmetry and divide the 24-bit state into two equal 12-bit states. After performing the cycling experiment on this input, we find that the number of cycles remains the same, and the bits of the intermediate states also remain the same as one half of the 24-bit state. The output of the 12-bit input in a cyclic structure is shown in Figure 5.6. As this 12-bit state is still symmetric (now only lengthwise), we can further divide it into two 6 bit states and try the same experiment again, the result of which is visible in Figure 5.7. We notice again that the number of states that form the cycle remains the same, and the bits in the intermediate states also do not change. Thus we can conclude that the initial 24-bit version derives the same results as four separate 6-bit inputs.

Another method to reach this conclusion is to view the 24-bit state as a lengthwise symmetric state, and remove one half of the state to make it a 12-bit state as in Figure 5.8. This state exhibits transversal symmetry, and can be further divided into the same 6-bit state as we saw in Figure 5.7.

From these observations, we can conclude that that the cycles formed by symmetric states form a partition of all the cycles formed. This explains why we decided to remove the symmetric states in the first place - it is a subset of the possible cycles for that input, and implementing it more than once makes it redundant as it has already been covered by the smaller states with no symmetry.

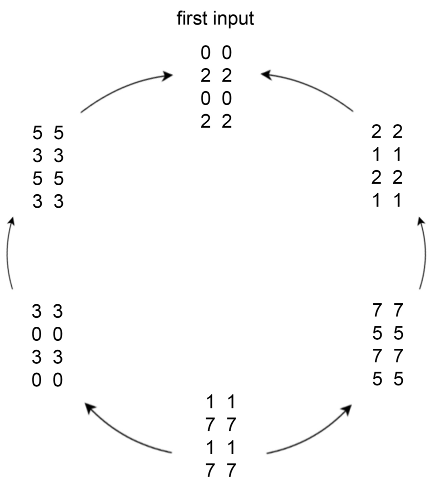

Figure 5.5: The cycles of a 24-bit state with symmetry

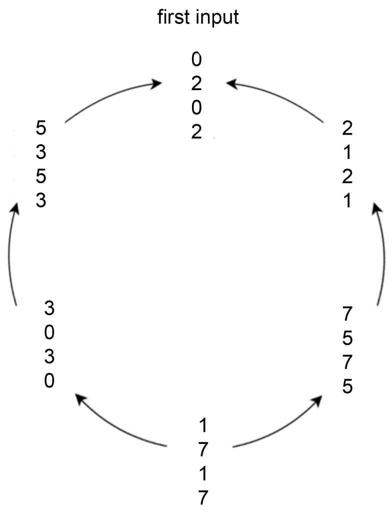

Figure 5.6: The cycles of the 12 bit state with symmetry

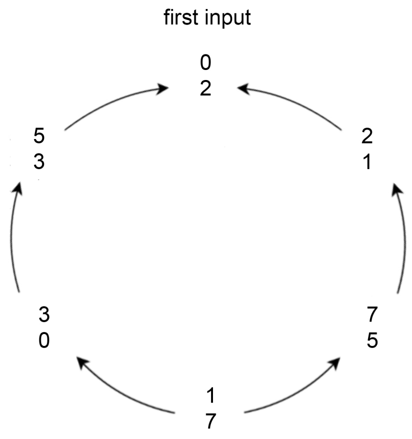

Figure 5.7: The cycles of the 6 bit state

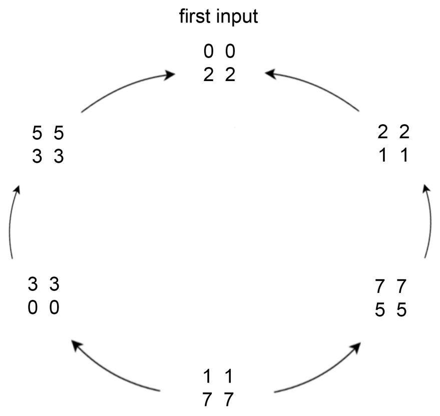

Figure 5.8: The cycles of the 12 bit state (dividing lengthwise)

Chapter 6

Results

In this chapter, we tabulate and explain the results of our implementation of the round function. 19 symmetry classes were considered and the round function implementation was performed on all possible inputs of each symmetry class. The remaining 18 symmetry classes are cumbersome and computationally intensive, and so we analyze the 19 classes that are considered in this report, and generalize the results for the remaining symmetry classes as well. For simplicity, we divide the symmetry classes based on the state size. Each section in this chapter elaborates on the results of all symmetry classes of a specific state size starting from the smallest, and moving on towards the largest. The possible symmetry within the state and shifted versions of the state are also illustrated in each section. The results that were derived from the implementation are tabulated for each symmetry class. Note that in this section, a cycle length of l includes each state that is involved in the cycle (including the initial input state) only once. We report on the cycle structure for each symmetry class, including the cycle lengths, number of cycles, and super-cycle count, super-state count. For better understanding, we define the cycle count and the super-cycle counts below as definitions.

Definition 9. *The super-cycle count refers to the total number of super-cycles formed by the symmetry class.*

Definition 10. *The cycle count refers to the total number of cycles formed by all the super-cycles of the symmetry class.*

The information on the cycles and cycle lengths help us form observations and calculations on the behaviour of the symmetry classes. We try to conclude whether or not the cycle structures resemble that of a random permutation. We also check if there are any patters that occur during the formation of cycles. A hypothetical example for a pattern would be if a specific string of three bits, say 010 always provides the same output, say 110 even if it is positioned in different planes of different states with varying dimensions. Thus, each cycle length, each super-cycle count, and each cycle count are important measures for us to estimate the cycle structure, and the behaviour of the symmetry class. The state count is not used to understand the cycle structure, but rather is used to check if our theoretical explanation and understanding of the total number of states matches the experimental results. This ensures that our implementation and experiments are not incorrect.

6.1 State size 3

Two symmetry classes with different basis vectors form states of size 3. When the input is a 3-bit block, each block or state is constructed using one bit per column, and similarly produces an output of 3 bits of the same structure. As the size of the input/output is 3 bits, there can be $2^3 = 8$ possible input/output permutations. The two inputs where all the bits are identical form one symmetry class and the remaining six inputs form the other symmetry class, both with basis vectors $\langle(1,0),(0,1)\rangle$.

When the size of the input is as small as three bits per state or one bit per plane, the concept of lengthwise, transversal, and skew symmetry do not exist. This is because there is only one bit per plane, and a single bit cannot be symmetric to anything. It is also not possible to shift bits in any direction with one bit per plane. Thus there will be no shifted versions of the state either. The results of the cycle experiments are summed up in Table 6.1 below.

We notice that in this case, the maximum cycle length is two. The first and last possible inputs that are., an integral input of 0 and 7 cause cycles length of one, meaning that for this symmetry class, applying the naive round function on the input results in the same output. The second symmetry state with six input states between the integral input 1 and 7 all form cycles with two states each. Together, the two symmetry classes form a total of five cycles, thus causing an odd parity when the state size is 3. For a better understanding of the 3-bit states, we have visualized all five cycles that are formed by these two symmetric states in Figure 6.1. The octal representation of the input value is written into each state.

Table 6.1: Results of cycle experiments on 3-bit Xoodoo

Cycle length	**States encountered**
1	1
1	1
2	2
2	2
2	2
5 cycles	8 states

6.2 State size 6

Three symmetry classes contain states of size 6 bits each. This state is made of three planes again, but this time with two bits in each plane. Two out of the three symmetry classes have basis vectors (1,0), (0,2) and (2,0), (0,1). There is also a third symmetry class that is made of basis vectors (1,1), (0,2). This rectangular grid structure is similar to the first grid mentioned, which is (1,0), (0,2). The only difference is the position of the bits in the structure and the order in which they are inserted and manipulated. This refers to both the initial insertion of the bit string into the representative state and

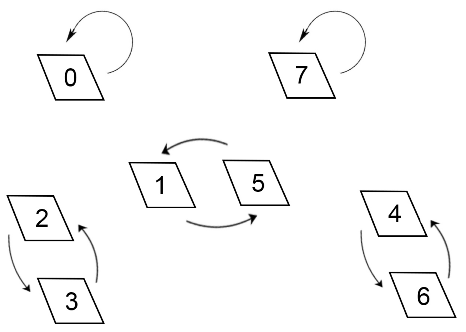

Figure 6.1: Cycles formed by the 3-bit states

the shifts involved in the different functions. Thus although these differ in position in terms of graphical visualization, during implementation the main difference between these states is the changes in shift values in the θ and ρ functions.

The 6-bit input with 2 bits per plane can exhibit lengthwise or transversal symmetry, but it can never exhibit skew symmetry. The $2x1$ version can be split vertically for symmetry, and the $1x2$ version can be split horizontally for symmetry. This can be seen in Figure 6.2 below. a stands for any number from the octal numbering system, and the plane indeed represents the entire state. There can also be only two shifted states for a 6-bit input, one is the identity state, and another is when the state is shifted by one bit either horizontally or vertically.

An input of 6 bits can have a total of 2^6 different states ranging from 000000 to 111111. Tables 6.2, 6.3 and 6.4 below provide the length of cycles formed in a super-state set (or a super-cycle) for the three symmetry classes discussed above. The length of the cycle is provided, along with the total number of cycles required to cover all states in the super-state set, which is labeled as count. The total number of cycles indicates the number of cycles required to cover all input states with an exception of the symmetric states.

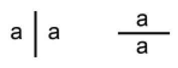

Figure 6.2: Transversal symmetry (left) and lengthwise symmetry (right) in a 6-bit Xoodoo state

Consider the symmetry classes that have states with a width of 1 bit, and a length of 2 bits. It can be noted that the two symmetry classes produce similar results, but a different number of cycles. The third symmetry class produces the most number of cycles. Also, all cycles are of odd parity. We can also notice that the number of states that are covered is 28, out of a total of 64 possible states.

Table 6.2: Results of cycle experiments on a 6-bit input with base vectors $\langle(1,0)(0,2)\rangle$

Base vectors	**Super-cycle length**	**Cycles per super-cycle**
(1,0) (0,2)	6	2
	6	2
	6	2
	4	1
	3	1
	3	1
		6 super-cycles
	28 super-states	9 cycles

Table 6.3: Results of cycle experiments on a 6-bit input with base vectors $\langle(1,1)(0,2)\rangle$

Base vectors	**Super-cycle length**	**Cycles per super-cycle**
(1,1) (0,2)	6	2
	6	2
	6	2
	4	1
	3	2
	3	2
		6 super-cycles
	28 super-states	11 cycles

Table 6.4: Results of cycle experiments on on a 6-bit input with base vectors $\langle(2,0)(0,1)\rangle$

Base vectors	**Super-cycle length**	**Cycles per super-cycle**
(2,0) (0,1)	6	2
	4	2
	4	2
	4	2
	3	1
	2	2
	2	2
	1	1
	1	1
	1	2
		10 super-cycles
	28 super-states	17 cycles

6.3 State size 12

The third version that this report examines uses 12 bits as input. Hence keeping the three planes intact, there are 4 bits in each plane. 4 bits can be visualized into three different structures which are a horizontal rectangle, a vertical rectangle, and a square. As seen in Chapter 4.3, a 12-bit state size has seven symmetry classes, two with width and length coordinates as 4x1, two with 2x2, and three symmetry classes with states of coordinates 1x4.

The three different structures of the 12-bit input can display different symmetries. The symmetry classes that have states of dimensions 1x4 and 4x1 are structured similar to the 6-bit states that we saw in the previous section, thus having the same symmetry properties as shown in the figure below. The 2x2 bit version is a bit more interesting - as it is a square, out of the three possible symmetries, it can display either one or all three at the same time. This is, of course, possible only in one case - when all the four bits in each plane are the same. This also can be viewed in Figures 6.3 and 6.4 below. Note that the values *a* and *b* can indicate any number from the octal numbering system, and the block represents a state of Xoodoo12.

A 12-bit input with 4-bit planes have four shifted versions, and these can be understood from Figure 6.5. The results of the experiment are tabulated into three tables. The results are divided in terms of structure. Tables 6.5, 6.6, 6.7, and 6.8 contain the results of the four symmetry classes of the input states with input dimensions of length 4 bits and width 1 bit. Tables 6.9 and 6.10 sum up results of the two symmetry classes with states of length 2 bits and width 2 bits, and finally, Table 6.11 contains outputs of the symmetry class with states made of length 1 bit and width 4 bits. From the results, we can see that the cycles are found to be different for each of these seven symmetry classes, the highest number of cycles being from the last symmetry class. Note that for Table 6.11, the column 'Multiplicity' refers to the number of such cycles with the given cycle length.

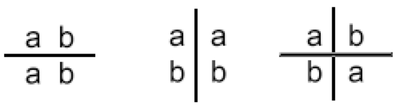

Figure 6.3: Lengthwise symmetry (left), transversal symmetry (center) and skew symmetry (right) in a 2x2 input block

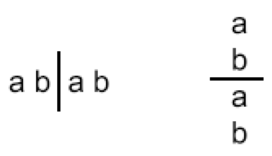

Figure 6.4: Transversal symmetry in a 1x4 input block (left) and lengthwise symmetry in a 4x1 input block (right)

Table 6.5: Results of cycle experiments on a 12-bit input with base vectors $\langle(1,0)(0,4)\rangle$

Base vectors	**Super-cycle length**	**Cycles per super-cycle**
(1,0) (0,4)	438	1
	302	2
	91	1
	83	1
	46	1
	41	1
	7	1
		7 super-cycles
	1008 super-states	8 cycles

Table 6.6: Results of cycle experiments on a 12-bit input with base vectors $\langle(1,1)(0,4)\rangle$

Base vectors	**Super-cycle length**	**Cycles per super-cycle**
(1,1) (0,4)	604	1
	277	4
	60	2
	35	1
	11	4
	9	2
	5	4
	3	4
	2	1
	1	4
	1	1
		11 super-cycles
	1008 super-states	28 cycles

Table 6.7: Results of cycle experiments on a 12-bit input with base vectors $\langle(1,2)(0,4)\rangle$

Base vectors	**Super-cycle length**	**Cycles per super-cycle**
(1,2) (0,4)	118	2
	105	2
	72	2
	66	2
	65	2
	60	2
	58	1
	49	2
	41	4
	38	1
	36	1
	35	2
	32	4
	22	4
	22	2
	22	4
	20	4
	19	2
	16	1
	13	2
	12	4
	12	1
	10	2
	10	4
	6	2
	5	2
	5	1
	5	2
	5	1
	4	1
	4	4
	4	1
	4	1
	4	4
	3	1
	2	2
	2	2
	1	4
	1	1
		39 super-cycles
	1008 super-states	86 cycles

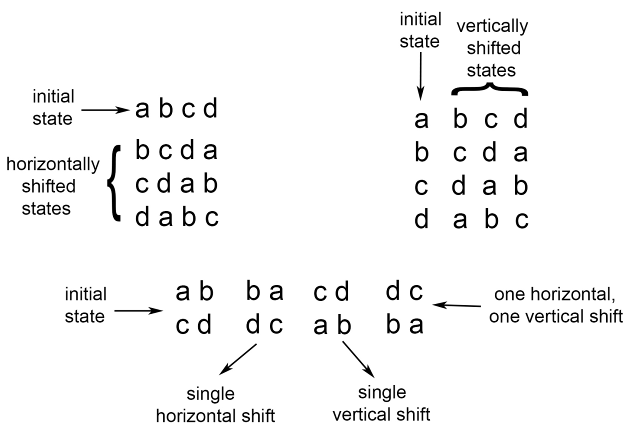

Figure 6.5: Shifted versions of a 4x1 input block (top-left), 1x4 input block (top-right), and 2x2 input block (bottom)

Table 6.8: Results of cycle experiments on a 12-bit input with base vectors $\langle(1,3)(0,4)\rangle$

Base vectors	**Super-cycle length**	**Cycles per super-cycle**
(1,3) (0,4)	435	1
	233	1
	224	1
	94	1
	10	2
	6	1
	2	1
	2	1
	2	1
		9 super-cycles
	1008 super-states	10 cycles

Table 6.9: Results of cycle experiments on a 12-bit input with base vectors $\langle(2,0)(0,2)\rangle$

Base vectors	**Super-cycle length**	**Cycles per super-cycle**
(2,0) (0,2)	474	2
	474	2
	18	2
	7	2
	7	4
		5 super-cycles
	980 super-states	12 cycles

Table 6.10: Results of cycle experiments on a 12-bit input with base vectors $\langle(2,1)(0,2)\rangle$

Base vectors	**Super-cycle length**	**Cycles per super-cycle**
(2,1) (0,2)	294	4
	294	4
	71	4
	71	4
	60	4
	36	4
	17	2
	17	2
	16	2
	16	2
	12	4
	12	4
	8	2
	8	2
	6	2
	6	2
	5	2
	5	2
	4	2
	4	4
	4	4
	3	4
	3	2
	3	2
	2	2
	2	2
	1	2
		27 super-cycles
	980 super-states	76 cycles

Table 6.11: Results of cycle experiments on a 12-bit input with base vectors $\langle(4,0)(0,1)\rangle$

Base vectors	**Super-cycle length**	**Cycles per super-cycle**	**Multiplicity**
(4,0) (0,1)	26	2	2
	24	1	4
	24	4	2
	22	2	1
	19	1	2
	18	1	4
	18	2	1
	16	4	2
	15	1	1
	14	2	8
	14	1	2
	12	1	4
	12	4	1
	12	2	2
	11	2	1
	11	4	1
	11	1	2
	10	4	4
	10	2	8
	10	1	2
	8	2	2
	8	1	2
	7	4	1
	7	2	1
	6	2	12
	6	4	2
	5	4	1
	5	2	1
	4	4	4
	4	1	4
	3	1	1
	3	4	1
	2	2	4
	2	4	4
	1	1	4
	1	4	3
	1	2	3
	1008 super-states	105 super-cycles 230 cycles	

6.4 State size 24

The 24-bit input block, just like the 12-bit version has a total of 7 symmetry classes. Each plane in a 24-bit input is made up of 8 bits. The way these bits are distributed into the planes depends on the basis vectors of the symmetry classes. One symmetry class contains states that are made up of planes with width 4 bits and length 2 bits, there are two symmetry classes two with planar coordinates 2x4, and three symmetry classes with width and length 1 bit and 8 bits respectively.

For a state where the plane has a width of 4 and length of 2, there is one identity state, and three equivalent planes modulo horizontal shift of value 1. Apart from this, there is also a vertical shift, which inverts the state in an upside-down manner. There are also two diagonal shifts, which consist of one horizontal and one vertical shift, one diagonal shift for each possible direction in which the shift can take place. Last but not least, there is also a shift that consists of one vertical and two horizontal shifts. Thus, there can be a total of 8 shifts for this specific symmetry class. The two other symmetry classes also display similar shifts with similar logic. Only, the number of vertical/horizontal shifts change. The shifts for the 2x4 version would be exactly the opposite of the 4x2 version, whereas the 1x8 version would simply be made up of the identity state, and seven vertical shifts. The shifts for these two symmetry classes are also shown in Figures 6.6, 6.7, 6.8, and 6.9. For each of these states again, symmetry is possible. For a 24-bit state, we can have lengthwise, transversal, and skew symmetry. This can be seen in Figures 6.10, 6.11, and 6.12.

Upon running the state space of the seven symmetry classes against our implementation, we derive results including the cycle count, cycle length, super-cycle count, and the total number of cycles and the total number of states that were covered. These results of the four symmetry classes with states of width 1 bit and length 8 bits are accumulated into Tables 6.12, 6.13, 6.14 and 6.15. Tables 6.16 and 6.17 contain results of the two symmetry classes whose states are made of width 2 bits and length 4 bits. The final Table 6.18 contains results of the other symmetry class where planes have a width of 4 bits and length of 2 bits. We can see that in all cases, the cycle structure parity is even.

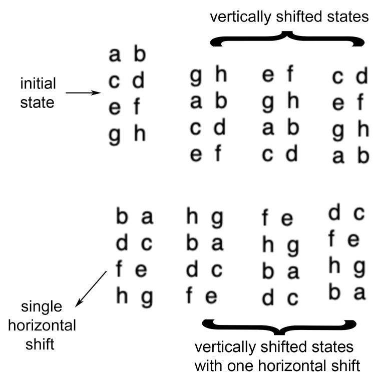

Figure 6.6: Shifted versions of a 2x4 input block

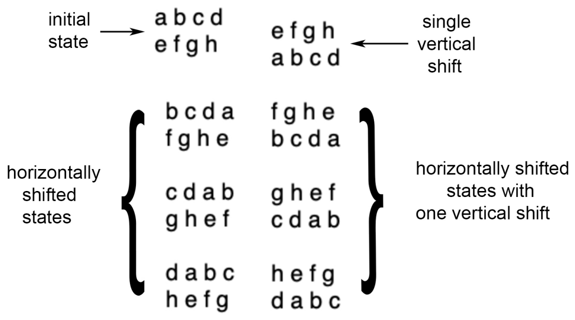

Figure 6.7: Shifted versions of a 4x2 input block

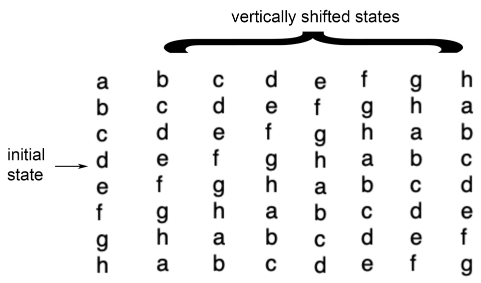

Figure 6.8: Shifted versions of a 1x8 input block

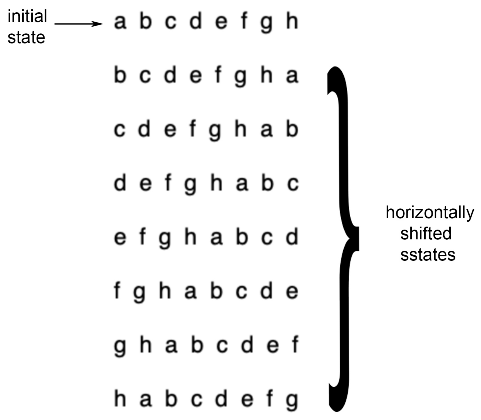

Figure 6.9: Shifted versions of a 8x1 input block

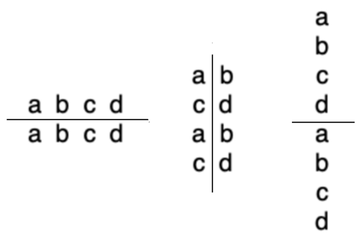

Figure 6.10: Lengthwise symmetry in a 24-bit state

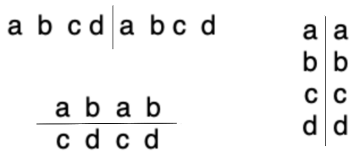

Figure 6.11: Transversal symmetry in a 24-bit state

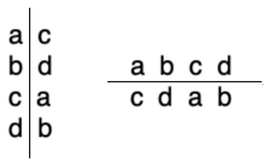

Figure 6.12: Skew symmetry in a 24-bit state

Table 6.12: Results of cycle experiments on a 24-bit input with basis vectors $\langle(1,0)(0,8)\rangle$

Base vectors	**Super-cycle length**	**Cycles per super-cycle**
(1,0) (0,8)	930680	2
	390124	1
	305741	1
	300922	1
	147453	1
	8924	4
	8117	1
	4039	2
	375	1
	126	8
	74	8
	59	1
	4	2
	2	1
		14 super-cycles
	2096640 super-states	34 cycles

Table 6.13: Results of cycle experiments on a 24-bit input with basis vectors $\langle(1,2)(0,8)\rangle$

Base vectors	**Super-cycle length**	**Cycles per super-cycle**
(1,2) (0,8)	1148852	2
	723997	8
	147616	2
	41688	8
	20330	4
	9824	1
	3331	2
	298	1
	279	2
	203	2
	126	8
	40	1
	28	4
	21	1
	6	1
	1	1
		16 super-cycles
	2096640 super-states	48 cycles

Table 6.14: Results of cycle experiments on a 24-bit input with basis vectors $\langle(1,4)(0,8)\rangle$

Base vectors	**Super-cycle length**	**Cycles per super-cycle**
(1,4) (0,2)	936428	2
	919253	2
	177511	1
	45916	1
	5481	4
	4861	1
	3962	4
	3095	4
	71	2
	39	8
	15	2
	8	1
		12 super-cycles
	2096640 super-states	32 cycles

Table 6.15: Results of cycle experiments on a 24-bit input with basis vectors $\langle(1,6)(0,8)\rangle$

Base vectors	**Super-cycle length**	**Cycles per super-cycle**
(1,6) (0,8)	2077506	4
	15181	1
	1847	1
	1422	1
	295	1
	191	1
	136	2
	34	4
	11	1
	11	2
	2	2
	2	2
	1	1
	1	1
		14 super-cycles
	2096640 super-states	24 cycles

Table 6.16: Results of cycle experiments on a 24-bit input with basis vectors $\langle(2,0)(0,4)\rangle$

Base vectors	**Super-cycle length**	**Cycles per super-cycle**
(2,0) (0,4)	1175312	2
	565503	8
	226827	8
	46258	2
	34650	2
	23884	4
	13082	2
	6435	2
	2146	4
	1266	4
	216	2
	44	4
	7	4
	2	4
		14 super-cycles
	2095632 super-states	48 cycles

Table 6.17: Results of cycle experiments on a 24-bit input with basis vectors $\langle(2,2)(0,4)\rangle$

Base vectors	**Super-cycle length**	**Cycles per super-cycle**
(2,2) (0,4)	1803216	4
	133452	2
	122413	2
	18774	8
	12582	2
	1989	2
	1450	2
	1033	8
	569	4
	54	2
	46	4
	25	2
	17	8
	6	4
	4	2
	2	4
		16 super-cycles
	2095632 super-states	32 cycles

Table 6.18: Results of cycle experiments on a 24-bit input with basis vectors $\langle(4,0)(0,2)\rangle$

Base vectors	**Super-cycle length**	**Cycles per super-cycle**
(4,0) (0,2)	1348667	2
	321293	1
	202166	1
	88299	8
	86948	1
	44469	8
	1163	1
	711	1
	686	1
	318	2
	298	8
	163	1
	124	1
	85	1
	76	2
	60	1
	42	1
	19	1
	12	2
	8	8
	5	1
	3	1
	3	1
	3	4
	2	8
	2	4
	2	4
	2	2
	2	8
	1	1
		30 super-cycles
	2095632 super-states	86 cycles

Chapter 7

Observations

This chapter speaks in detail about the observations and findings from the results of the experiments that were gathered in the previous chapter. These observations include those that are specific to each symmetry class, some observations that are generic to given state size, and those that are generic to all symmetry classes and Xoodoo as a whole.

7.1 The θ function

Consider the θ function step mapping for an state of size 3. The θ-effect or the E plane in this scenario is always 0. This is caused as a result of performing an XOR on the same plane twice in the previous step. XOR of two identical bits results in a 0 bit. Thus, the second step of the θ function always results in 0. This further makes the next step produce the same output as the initial state. Thus the θ function becomes an identity function.

If we look into the 6-bit state, we have two possible structures. In the case of $2x1$ state, we notice that the θ function, just like in a 3-bit construction becomes an identity function. This is because the second step leads to E always equal to 0, and the θ function produces the same state as its output as well.

This property can also be noticed in symmetry classes with states of state dimensions of width 4 bits and length 1 bit, and also the symmetry classes with dimensions width 8 bits and length 1 bit. We can hence generalize the following in a conjecture.

Conjecture 1. For all symmetry classes that contain states with length as 1, the θ function becomes an identity function.

7.2 The ρ and χ functions

If the θ function becomes an identity, then the naive Xoodoo round function only executes the ρ steps and the χ step. For the case of the 3-bit symmetry classes, the ρ_{west} and ρ_{east} steps can again be ignored, as it is not possible to perform a shift on a one-bit plane. This simplifies the algorithm and brings it down to just the χ function. Furthermore, if the state is such that all three planes contain the same bit value ie., if the input to the state is 000 or 111, then the χ function also becomes an identity.

If we consider symmetry class with an state size of 6 bits with a length of 1 bit, then the $\rho_e ast$ function disappears as there cannot be any shifts in the z-direction, and the x-direction shifts modulo 2 are always 0. ρ_{west} acts as an identity when the second plane out of the three are made of the same two bits - either 0 and 0, or 1 and 1. The only function that can never be an identity is the χ function. In such cases where the ρ function also disappears, only the χ function executes. This simplifies the execution, but could also lead to a weaker bit interaction. A weaker bit interaction causes shorter cycles, thus increasing the total number of cycles.

Conjecture 2. When the θ and ρ functions collapse, the only remaining step in the naive Xoodoo round function is the χ function, and this leads to weak bit interaction.

7.3 Correlation between the state dimensions and the super-cycle count

As we saw in the first section of this chapter, some symmetry classes cause the θ function to become an identity, thus limiting the interaction between the bits. In this section, we try to understand whether and how this is affected by the width and the length of the planes in a state. For this, rather than looking at the total cycle count, we have taken into consideration the number of super-cycles of the state. It is visible from Table 7.1 that as the length decreases, the number of cycles steadily increases. To understand this further, we have created a line graph (refer to Figure 7.1) using the average cycle count for each possible width of each symmetry class, and plotted it against the width itself. The symmetry classes are organised with respect to the state size. Beginning at the lowest possible width value, this graph displays a steady decrease in the number of cycles as the width increases for three set of symmetry classes. We can notice that the number of cycles is initially very high, but they decrease to the expected amount. This could be accounted for by the fact that the θ function disappears when the width is 1, and as the width steadily decreases from the highest possible value, the bit interaction within the lanes of the state also are limited. We can hence find a direct correlation between the width of the state and the number of cycles formed. The symmetry class with basis vectors $\langle(4,0)(0,1)\rangle$ that has a width of 1 bit for a 12-bit state is an anomaly case, where the interaction is even more limited due to the reduced width and increased length.

Conjecture 3. As the length increases and the width decreases, the θ function slowly disappears, and the number of cycles increases.

7.4 States per cycle

For all the symmetry classes, we evenly notice that the total number of states that were involved in the formation of cycles remained the same for a constant state size. A table showing the input size and the total number of states is tabulated in Table 7.2 below.

Table 7.1: Total number of super-cycles for different symmetry classes

Input size	**Basis vectors**	**Length**	**Width**	**Cycle count**
3	(1,0) (0,1)	1	1	5
6	(1,1) (0,2)	1	2	6
	(1,0) (0,2)	1	2	6
	(2,0) (0,1)	2	1	10
12	(1,0) (0,4)	1	4	7
	(1,3) (0,4)	1	4	9
	(1,1) (0,4)	1	4	11
	(1,2) (0,4)	1	4	39
	(2,0) (0,2)	2	2	5
	(2,1) (0,2)	2	2	27
	(4,0) (0,1)	4	1	105
24	(1,4) (0,8)	1	8	12
	(1,0) (0,8)	1	8	14
	(1,6) (0,8)	1	8	14
	(1,2) (0,8)	1	8	16
	(2,0) (0,4)	2	4	14
	(2,2) (0,4)	2	4	16
	(4,0) (0,2)	4	2	30

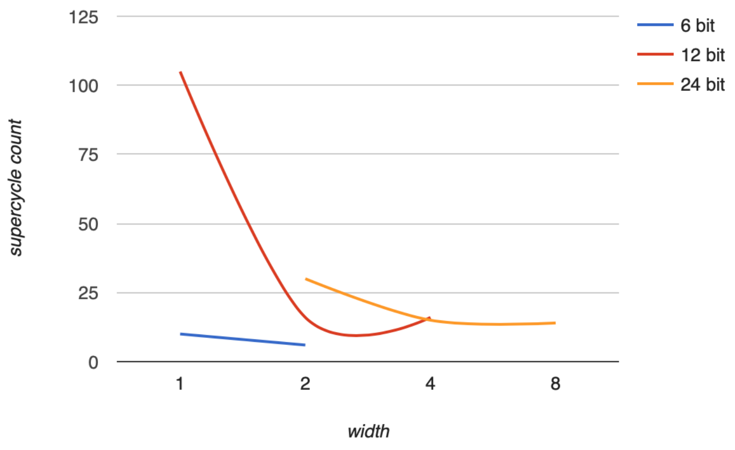

Figure 7.1: A line graph of state width vs the super-cycle count

Table 7.2: Input version and the total number of states covered

Input size	**Number of states**
3	8
6	28
12	980, 1008
24	2096640, 2095632

7.4.1 3 bit state

The 3-bit state does not display symmetry within the state, nor can it have shifted versions as there is only one bit per plane. There are a total of $2^3 = 8$ states, and thus all 8 states are involved in the cycle. These are the basic cycles that will later be a part of versions of all other state sizes.

7.4.2 6 bit state

A 6-bit state consists of 2-bit planes. The number of possible states are calculated as $2^{3*2} = 64$. The number of symmetric states is $2^{3*1} = 8$ states in total. This value is subtracted from the total number of states as we ignore the symmetry states. Thus getting 56 states. We also remove the remaining states from the super-state set, therefore dividing 56 by the number of states in the super-state set. We then get a total of 28 states.

7.4.3 12 bit state

The 12-bit state can take three different figures - a one-dimensional array, and a square. We consider both these scenarios below.

- An state of 12 bits can have a total of $2^{12} = 4096$ possible states. The number of symmetric states would be $2^6 = 64$ states if it is in the form of a one-dimensional array. As a first step, we subtract these symmetry states from the total. This gives 4032 states. This is now divided by the number of super-states in a set, which is four. This gives us a total of 1008 states.

- The second scenario is more tricky, as we are dealing with a more complex structure than a one-dimensional array. A square can have three types of symmetries - horizontal, vertical, and diagonal. These should be subtracted from the total number of states. This is calculated as $4096 - 3 * 64 = 3904$ states. But we should also consider the fact that all three cases include some states in common - the ones where all bits are zero, or all bits are one. The probability that all bits are zero or all bits are one in a plane would be 1, but if we consider three planes that make up the state, this value would be 2^3. This value should be added twice to make up for it being subtracted thrice for the three types of symmetry. Thus we have $3904 + 16 = 3920$. If this amount is divided by four for representing the super-state sets, then we have a total of 980 states.

7.4.4 24 bit state

For the 24 bit state, there are two possibilities. Let us say we have a state that is made of three planes, each containing n bits. They are arranged in a two-dimensional array of dimensions x by z. Let us say n is a power of two such that $n = 2^e$, where e is an integer. For an state of 24 bits, we can have two scenarios. These are explained below.

- If the state is a one-dimensional array, meaning either the x-coordinate or the z-coordinate is 1, then symmetry can only be either horizontal or vertical. The number of symmetric states, that is., the number of states that have two halves is $2^{3*2^{e-1}}$. This is calculated by finding the probability that an eight-bit plane has two halves. There are n bits per plane, so if we fix $n/2$ of the bits in three planes, we have a total of $3*n/2$ bits that can take values either 0 or 1. So the total number of states would be $2^{3*n/2}$. But as we take n to be of the form 2^e, we can simplify the formula to get $2^{3*2^{e-1}}$. These states are ignored during the implementation. So, we can remove them from our count by subtracting them from the total number of possible states to find the number of non-symmetric states as $2^{3*2^e} - 2^{3*2^{e-1}}$. As we know, there are a total of n super-states in a super-state set. So we can divide this by 2^e to get the following formula to achieve 2096640 states as $(2^{3*2^e} - 2^{3*2^{e-1}})/2^e$.

- If both the x and z coordinates are greater than one, then there can be three types of symmetric states, namely horizontal, vertical, and diagonal. Each of these has sizes $2^{3*2^{e-1}}$ as calculated before. This value has to be subtracted from the total number of states thrice, once for each symmetry. But it has to be taken into consideration that there also exists a subset of periodic states that consists of four equal sub-states arranged in a square, so these states have to be added again. This value must again, further be divided by the number of states in a super-state set, which is 8, and thus we reach a total of 2095632 states.

7.5 Expected number of super-cycles

For a random permutation, the number of cycles can be calculated as the following [17].

$$\text{Number of cycles} = ln(2) * inputSize$$

For all inputs other than the 3-bit input, we consider the super-state sets, so we must keep this in mind during our calculation. For a 6-bit input, the state space becomes $2^6/2$ as there are 2 states per superset. So, the input size is considered as 5 bits rather than 6 bits. The state space for a 12-bit input is $2^{12}/4$ as the super-state set contains four states each, making the input size as 10 bits instead of 12 bits. Similarly, for the 24-bit input, it is $2^{24}/8$, thus making the input size as 21 bits instead of 24 bits. We calculate the cycles for these modified input sizes using the formula given above, but they are actually super-cycles of the original input sizes.

A table with expected number of cycles for a random permutation of the given input size is provided below (see Table 7.3). The expected number of cycles is not exact, it is rounded off to the closest integer. The next two columns display the lowest and highest super-cycle count out of all possible symmetry classes for the respective input size. The last column shows the average number of super-cycle for the specific input size, and it is compared with the expected cycle length. We see that the results do not match the expected count in all cases. This is because the formula that is used here is applicable for relatively large domains, Therefore, while dealing with small inputs of size 3, 6, and 12 bits, we cannot say that the formula will apply perfectly. This is also another opportunity to understand how the bit interactions affect the cycles count. The average accounts for all symmetry classes, including the ones with an excessively high number of cycles. With a very low width, the anomaly case of 105 cycles contributes

Table 7.3: Expected number of cycles for a random permutation versus actual number of super-cycles for Xoodoo

Input size	**Expected number of cycles**	**Super-cycles (lower bound)**	**Super-cycles (upper bound)**	**Super-cycles (average)**
3	2	5	5	5
6	4	6	10	7
12	7	5	105	29
24	15	12	30	16

to the 12-bit states having a very high average. We can also notice that for the highest input considered here, which is the 24-bit input, the actual cycle length is closest to the expected. Therefore as the input increases and tends to infinity, the formula is more relatable. Conjecture

Conjecture 4. The average number of super-cycles from of all the symmetry classes of Xoodoo with states of a given size *n* moves closer to the expected number of cycles for a random permutation of size *n* as the input size increases.

Table 7.4 divides the super-cycle counts in an ascending order of state length. The average super-cycle count of all symmetry classes with a given state size and a given length is compared against the expected number of cycles. We notice that for a fixed state size, as the length of the state increases, the number of super-cycles moves closer to the theoretically expected number of cycles. In other words, as *z* increases, the cycle length reduces and stabilizes. We write this as a conjecture. Conjecture

Conjecture 5. Let us consider a set of symmetry classes with states of a given size *n*. As the length of the state increases, the super-cycle count tends to the number of cycles that a random permutation of the same size is expected to produce.

7.6 Expected cycle lengths

Of all the cycles possible, it is known that the longest cycle covers roughly 60% of the entire permutation. Research by the mathematician Golomb and few others involving the largest cycle in a random permutation resulted in the Golomb-Dickman constant[15]. If a_n is the average of the lengths of the longest cycles from every possible permutation of a set of given size n, then the Golomb-Dickman constant is calculated as follows. [23]

$$\lambda = \lim_{n\to\infty} a_n/n$$

This value is calculated to be $\lambda = 0.6243299...$. Below is Table 7.5 showing the longest cycle of each symmetry class of each input size that we saw in the previous chapter. We notice that these results are not always close to what is expected from theory, again due to the size of our inputs. As we see from the formula, this value can be

Table 7.4: Expected number of cycles for a random permutation versus actual number of super-cycles for Xoodoo

Input size	**Basis vectors**	**State length**	**Super-cycle count**	**Super-cycles average**	**Expected number of cycles**
3	(1,0) (0,1)	1	5	5	2
6	(1,0) (0,2)	2	6		
6	(1,1) (0,2)	2	6	6	4
6	(2,0) (0,1)	1	10	10	4
12	(1,0) (0,4)	4	7		
12	(1,1) (0,4)	4	11		
12	(1,2) (0,4)	4	39		
12	(1,3) (0,4)	4	9	16	7
12	(2,0) (0,2)	2	5		
12	(2,1) (0,2)	2	27	16	7
12	(4,0) (0,1)	1	105	105	7
24	(1,0) (0,8)	8	14		
24	(1,2) (0,8)	8	16		
24	(1,4) (0,8)	8	12		
24	(1,6) (0,8)	8	14	14	15
24	(2,0) (0,4)	4	14		
24	(2,2) (0,4)	4	16	15	15
24	(4,0) (0,2)	2	30	30	15

computed precisely only in the cases where the domain size is very large, and the inputs considered in this report are relatively smaller.

7.7 Parity of the cycle structure

A permutation over a set with an even number of elements is even if and only if its cycle representation (including the fixed points) contains an even number of cycles[29]. Consider a permutation P over a set of even number of elements. If the elements are of size 48 bytes, then represent the permutation as a concatenation of 48 permutations that carry out transformations for one byte, and keep the remaining 47 bytes fixed. The concatenation of even permutations always yields an even permutation, thus we can establish that for an even number of elements, the number of cycles will always be even. Thus for all of our states with an exception of the 3-bit state, our experiments must result in an even number of cycles. The 3-bit state has an odd number of elements, and so it does not fall into this category. Table 7.6 summarises whether the input sizes have an even or odd number of cycles, followed by an explanation for the same.

We notice that although the 6-bit state must produce an even number of cycles, it produces an odd number of cycles. This is due to the symmetry states that we ignored

Table 7.5: Cycle lengths: expected vs actual

Input size	**Basis vectors**	**Actual maximum**	**Expected maximum**
3	(1,0) (0,1)	2	3
6	(1,1) (0,2)	6	16
	(1,0) (0,2)	6	16
	(2,0) (0,1)	6	16
12	(1,0) (0,4)	438	605
	(1,3) (0,4)	435	605
	(1,1) (0,4)	604	605
	(1,2) (0,4)	118	605
	(2,0) (0,2)	474	605
	(2,1) (0,2)	294	605
	(4,0) (0,1)	26	106055
24	(1,4) (0,8)	936428	1257984
	(1,0) (0,8)	930680	1257984
	(1,6) (0,8)	2077506	1257984
	(1,2) (0,8)	1148852	1257984
	(2,0) (0,4)	1175312	1257379
	(2,2) (0,4)	1803216	1257379
	(4,0) (0,2)	1348667	1257379

during implementation. This can be revived again be adding the cycles from the 3-bit state. Both 6 bit and 3-bit states result in an odd number of cycles, and therefore we add the odd and odd number of cycles to finally get an even number of cycles. The 12 bit and 24-bit states have an even number of cycles but are still missing the symmetric states of the state space. To find the actual parity of a cycle structure, the number of super-cycles counts of all states smaller than themselves must be added to its cycle count. Thus the 12-bit and 24-bit states remain to have even parity. Therefore, for a given state size, our implementation ignores all symmetric states that can be of smaller dimensions, but the total cycles are always even.

Table 7.6: Cycle count parity: expected versus actual

Input size	**Expected parity**	**Actual parity**
3	odd	odd
6	even	odd
12	even	even
24	even	even

7.8 Varying cycle counts for a given state size

With an exception of the simple 3-bit states, we compare the cycle structures of the symmetry classes with a given state size and different dimensions. We notice that the highest variation occurs when the state size is 12 bits. The super-cycles formed by symmetry class with basis vectors $\langle(2,0),(0,2)\rangle$ were a total of just 5, while for another symmetry class with basis vectors $\langle(4,0),(0,1)\rangle$, we established a total of 105 super-cycles. There could be various factors that influence and contribute to this cause. To understand this, we look into symmetry classes with smaller dimensional states. For the symmetry class with basis vectors $\langle(2,0),(0,1)\rangle$, we can notice that the interaction between the bits in the intermediate states of the cycle is more restricted than when compared to the interaction between the bits of the states of the symmetry class with basis vectors $\langle(1,0),(0,2)\rangle$. The cycle count of the former was also recorded to be more than the cycle count of the latter. The bits are recycled within the same bit positions and this results in a smaller cycle length, as it comes back to the original state sooner.

It can also be noted that when some steps in the naive Xoodoo round function become identity functions, the number of states in a cycle reduces. This can be specifically seen in the generic case where the length is just 1 bit. For every state size with the exception of the 3 bit state, we can see that the symmetry class with the highest width dimensions (which occurs when the dimensions of the length is at its lowest possible) also result in the highest number of cycles among all cycles produced by symmetry classes of the same state size. In the case of a 24-bit state, the number of cycles is a bit higher when the length of a plane is 2 bits and the width is 4 bits long. Upon investigation of some shorter cycles from the given variation of the 24 bit Xoodoo, it can be observed that the intermediate states recycle the same bit positions and there is not much confusion or diffusion. Therefore, due to the low interaction between bits among the x-coordinate, the total number of cycles is almost twice as what was expected. We make some final conjectures after analyzing all possible observations from the results: Conjecture

Conjecture 6. The disappearing of the θ and ρ functions leads to a decrease in the number of states in each cycle, and this in turn leads to a high number of cycles. The cycle structure is thus determined by the behaviour of certain step mappings of the naive Xoodoo round function.

7.9 Overview of observations

This chapter analyzed the results that we derived from our cycling experiments. We tabulated the results of 19 symmetry classes in Chapter 6, where we included information regarding the cycle count, super-cycle length, super-state count, and super-cycle count. We also saw that the parity of the cycle structure remains the same for a given state size, with no regards to the symmetry class or the dimensions of the state. Many observations were made from these results. The θ function is seen to slowly disappear as the length of the permutation reduces, and this is mentioned in the first section of this chapter, after which we moved on to the behavior of the remaining steps of the naive Xoodoo round function. We understood how the state size and its dimensions correlate with the super-cycle count and visualized this with a line graph. We calculated the number of states that were encountered for each symmetry class and found that our results matched the theoretical explanation for the same. The next sections compared the number of cycles and cycle lengths against expected values for the same and provided possible explanations for outliers. We then moved on to speak about the parity of the cycle structure and concluded the chapter with a discussion regarding the observation that for a given state size, the number of cycles is not always the same.

Chapter 8

Conclusion

This report has contributed to a unique understanding of the cipher by investigating the naive Xoodoo round function analyzing the cyclic structure of the cryptographic permutation Xoodoo. We learned about the invariant subspace attack that can be applied to lightweight block ciphers. The main goal of this thesis was to deduce the vulnerability of the permutation to such an attack. For this, cycling experiments were performed on the symmetry classes of Xoodoo by making use of an implementation of the round function while taking into account various concepts and properties that Xoodoo exhibits. The most important property that this research revolved around was the symmetry. It affected the behavior of the input when the round function was applied to it, and also led to some anomaly cases. The cycle lengths of the different symmetry classes of each of these inputs were tabulated and analyzed. The various cycle lengths of the permutation give us information regarding the interactions between the bits, and we can understand better the strength of the system. All the research questions that were introduced in Chapter 1 were addressed and answered. The conclusions of this thesis are listed below.

- Is Xoodoo vulnerable to the invariant subspace attack?

  We can notice that the results of the cycle structure of all the 19 symmetry classes that were analyzed during the course of this research are close to a random permutation. Evaluating the cyclic behaviour of the symmetry classes whilst keeping in mind that the size of the inputs were much smaller than what is generally expected for a random permutation, and also recognising the few anomaly cases, we can safely say that the cycles in Xoodoo do not reveal obvious patterns within them, and the number of cycles are in an acceptable range within the expected values. Thus we can establish that except for the symmetry class with basis vectors $<(4,0),(0,1)>$, which strays slightly from the expected cycle count of a random permutation, the invariant subspace attack is highly unlikely on the inputs of the permutation Xoodoo.

- How many cycles do different inputs produce, how and why is it different from what is expected, if at all?

  The cycle lengths are tabulated into Table 7.3. Our findings demonstrate that for a 6 bit and 12-bit input, the number of cycles is highest for the symmetry class where the length of the plane is simply 1 bit. For inputs of symmetry classes where the length cannot be 1 (that is, cases where the state size is 24 and above), the number of cycles is highest when the length is the least possible, and the width is the maximum possible. This is caused by weakened bit interaction among the rows of the planes and the state. We notice that the symmetry class with basis vectors $<(4,0),(0,1)>$ is again an anomaly case with a very high number of cycles.

- What are the parities of the cycle structures in the different symmetry classes and how is it so?

  The parity of the total cycle count is always even for a permutation with an even number of elements. For Xoodoo, the cycle count is even only when all the symmetry states are included and all input states are taken into consideration, or in other words, when the entire input space is taken into consideration. The parity of cycle structures with an state size of 3 is odd, and this is due to its odd state size. The property of symmetry allowed our implementation to be simplified for inputs larger than 3 bits. This caused the parity of the cycle structures with an state size of 6 bits to be odd as the super-states are taken into account. But, when the symmetric states are also included, the final cycle count sum would be even. Thus when we add the cycle count of the 3-bit cycles to the 6-bit cycles, then the parity of the 6-bit cycles becomes even. Furthermore, for 12-bit symmetry classes and 24-bit symmetry classes, the sum is again even. Therefore, we can conclude that the parties of the cycle structures are very much as expected, but without the super-states, we notice an interesting pattern of parity.

- How do the different steps of the round function behave with varying state sizes?

  We noticed cases where the number of rows in the planes of a state is equal to one. This led to the θ function disappearing, or in other words, the θ function becomes an identity. The bit manipulation and bit interaction are directly affected by the θ function, and thus the number of cycles drastically increases. We can say that as the interaction between the bits decrease, the cycle length decreases, and the total number of cycles increases. This could be a strong explanation for the symmetry class with basis vectors $<(4,0),(0,1)>$ having a very high count of 105 super-cycles.

- Describe the role that symmetry plays in this research.

  We examined the symmetry classes of Xoodoo, but not all of them. Some symmetry classes have states of very high state sizes, and the implementation of the same could take weeks, months, or even years due to its size complexity. Apart from being time-extensive, it would also require high amounts of computational power. The symmetry property in Xoodoo ensures that we can extend our conclusions to all versions of Xoodoo. Thus, finding similar patterns in smaller state sizes can also be used to derive conclusions for inputs of larger sizes. Therefore our results regarding the invariant subspace attack and others can be generalized for all symmetry classes and inputs.

- How do different symmetry classes vary in terms of cycle count?

  Two symmetry classes could have states with the same state size but with varying dimensions. We notice that the number of cycles depends strongly on the dimensions of the state. As the length decreases, the interaction between the bits of a plane also reduces, thus causing an increase in the number of cycles. The shifts that are involved in the step mappings of the round function are also different for different symmetry classes of the same state size. Thus, the cycle count is subject to change as the dimensions of the state are changed.

The limitation that we faced during this research was that for bigger sizes of the cipher, we required a drastically higher amount of computational power and time. But the concept of symmetry allows us to extend our results from the symmetry classes with smaller basis vector coordinates to the bigger versions as well. Thus, the observations and patterns that we notice in these versions will apply also to inputs of size 48, 96, and 128.

8.1 Recommendations for future work

Many different adaptations in shifts, tests with symmetry, and experiments regarding the cycle count have been left for the future due to lack of time and/or computational power. The possible future work to continue this research would concern deeper analysis of the bit interactions, implementing larger inputs and maybe in a more efficient way with the help of code parallelization, understanding how shifts affect the cycle count by altering them and analyzing output, and a few more. Some of these are listed below in more detail.

- Due to the constraint on time and computational power, inputs higher than 24 bits were not included in this report. There are three possible inputs beyond this 24-bit input, namely 48-bit input, the 96-bit input, and the 128-bit input. It would be interesting to see how the cycle lengths vary for these inputs and to see if the conclusions of this report still hold good for the two bigger state sizes as per symmetry.
- Digging deeper into the intermediate states of the cycles and performing further data analysis into the interaction of bits might also provide further results on how the bits are interacting, and why the cycle lengths differ so widely when the coordinates change. The weak interaction between the bits leads to a smaller cycle length, and this could give significant results.
- The symmetry class with basis vectors $\langle(4,0),(0,1)\rangle$ has an extremely high number of cycles: a total of 105. This stands out compared to the cycles produced by all other symmetry classes. The result that we derived has to with the fact that the length of the input permutation is low, and the width is high. Further investigation into this symmetry class would help to understand this anomaly case.

- The symmetry classes with basis vectors $\langle(2,0),(0,2)\rangle$ and $\langle(2,1),(0,2)\rangle$ contain more than one cycle of the same length. The total number of states and the parity of the cycle structure are backed by theory, but having more than one cycle of the same length is theoretically highly improbable. In-depth research and reasoning regarding how these cycles were formed could be a possible future work.

- We saw how the length and width affect the number of cycles produced. In addition to this, the cycle count might also be affected by the bit shifts. The higher the number of bit shifts in the x-direction and lower the number of bokit shifts in the z-direction, there could be an increase in the number of cycles. Although this cannot be proved theoretically, it would be interesting to understand if, how, and why this occurs.

- The current implementation is very time consuming, and some efficient code parallelization might make execution much faster. This is something that has not been performed within the scope of this thesis and would be an interesting experiment to perform. This experiment is different from all the others mentioned above as it does not try to answer any observation from the results of this research, but rather is about performing the same study on inputs that demand much more computational power and time.

Bibliography

[1] Hamdan Alanazi, B Bahaa Zaidan, A Alaa Zaidan, Hamid A Jalab, M Shabbir, Yahya Al-Nabhani, et al. New comparative study between des, 3des and aes within nine factors. *arXiv preprint arXiv:1003.4085*, 2010.

[2] Daniel J Bernstein, Stefan Kölbl, Stefan Lucks, Pedro Maat Costa Massolino, Florian Mendel, Kashif Nawaz, Tobias Schneider, Peter Schwabe, François-Xavier Standaert, Yosuke Todo, et al. Gimli: a cross-platform permutation. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 299–320. Springer, 2017.

[3] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 313–314. Springer, 2013.

[4] Guido Bertoni, Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. Farfalle: parallel permutation-based cryptography. *IACR Transactions on Symmetric Cryptology*, pages 1–38, 2017.

[5] Alex Biryukov and David Wagner. Slide attacks. In *International Workshop on Fast Software Encryption*, pages 245–259. Springer, 1999.

[6] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.

[7] Joan Daemen, Seth Hoffert, G Van Assche, and R Van Keer. The design of Xoodoo and Xoofff. *Transactions on Symmetric Cryptology*, pages 212–226, 2018.

[8] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654, 1976.

[9] PUB FIPS. Data Encryption Standard (DES). *National Institute of Standards and Technology*, 25(10):1–22, 1999.

[10] Burton S Kaliski, Ronald L Rivest, and Alan T Sherman. Is DES a pure cipher?(Results of more cycling experiments on des)(preliminary abstract). In *Conference on the Theory and Application of Cryptographic Techniques*, pages 212–226. Springer, 1985.

[11] Jonathan Katz, Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 1996.

[12] Lars Knudsen, Gregor Leander, Axel Poschmann, and Matthew JB Robshaw. Printcipher: a block cipher for ic-printing. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 16–32. Springer, 2010.

[13] Xuejia Lai. *On the design and security of block ciphers*. PhD thesis, ETH Zurich, 1992.

[14] Gregor Leander, Mohamed Ahmed Abdelraheem, Hoda AlKhzaimi, and Erik Zenner. A cryptanalysis of printcipher: the invariant subspace attack. In *Annual Cryptology Conference*, pages 206–221. Springer, 2011.

[15] Pieter Moree. Integers without large prime factors: from ramanujan to de bruijn. *arXiv preprint arXiv:1212.1581*, 2012.

[16] NIST. DES Archived Publication, 2005. URL `https://csrc.nist.gov/csrc/media/publications/fips/46/3/archive/1999-10-25/documents/fips46-3.pdf`.

[17] A.D. Barbour R. Arratia and S. TavarÃ©. *Logarithmic combinatorial structures: a probabilistic approach*. European Mathematical Society (EMS), ZÃŒrich,, 2003.

[18] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[19] Kenneth H Rosen and Kamala Krithivasan. *Discrete mathematics and its applications: with combinatorics and graph theory*. Tata McGraw-Hill Education, 2012.

[20] Bruce Schneier. Description of a new variable-length key, 64-bit block cipher (blowfish). In *International Workshop on Fast Software Encryption*, pages 191–204. Springer, 1993.

[21] Bruce Schneier. *Schneier's Cryptography Classics Library: Applied Cryptography, Secrets and Lies, and Practical Cryptography*. Wiley Publishing, 2007.

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

[23] LA Shepp and S_P Lloyd. Ordered cycle lengths in a random permutation. *Transactions of the American Mathematical Society*, 121(2):340–357, 1966.

[24] Nigel P Smart. *Cryptography made simple*, volume 481. Springer, 2016.

[25] Ko Stoffelen and Joan Daemen. Column parity mixers. *Transactions on Symmetric Cryptology*, pages 126–159, 2018.

[26] Wojciech A Trybulec. Basis of vector space. *Formalized Mathematics*, 1(5): 883–885, 1990.

[27] Walter Tuchman. Iv.hellman presents no shortcut solutions to the des'. *IEEE spectrum*, 16(7):40–41, 1979.

[28] G. Viravalli. Implementation of the naive xoodoo round function. `https://github.com/mengs7/Xoodoo-implementation`, 2019.

[29] Ralph Wernsdorf. The round functions of rijndael generate the alternating group. In *International Workshop on Fast Software Encryption*, pages 143–148. Springer, 2002.

[30] Wikipedia contributors. Invariant subspace — Wikipedia, the free encyclopedia, . URL `https://en.wikipedia.org/w/index.php?title=Invariant_subspace&oldid=909520786`.

[31] Wikipedia contributors. Invariant subset — Wikipedia, the free encyclopedia, . URL `https://proofwiki.org/wiki/Definition:Invariant_Subset`.