Computer Science Project

Pages 1–28
2023

Try different polynomials in the structure of MiMC and analyse efficiency, and basic cryptographic properties.

Simonas Tautvaišas
BSc. Mathematics and Computer Science
School of Mathematics
Supervisor: Dr Rishiraj Bhattacharyya
Student ID: 2204062
Words: 10299
2023-12-16

ABSTRACT

This paper delves into the development and analysis of novel cryptographic ciphers tailored to address modern cryptographic challenges, particularly in zero-knowledge proofs (ZK), secure multi-party computation (MPC), and fully homomorphic encryption (FHE). The focus of ciphers tailored towards these concepts lies on minimising the number of multiplications within Galois fields. In this project, MiMCGe class of cipher is created and an introductory cryptanalysis to argue their security is given. Although definitive proof of security for symmetric key ciphers is an open problem, the paper demonstrates the security of these ciphers against well-known attack paradigms. Moreover, the ciphers undergo rigorous testing to evaluate cryptographic properties such as diffusion, confusion, and randomness.

1 INTRODUCTION

In modern day cryptography, there are many new paradigms that focus on problems beyond classic confidentiality and authenticity between two-party communications. Research ideas like zero-knowledge proofs (ZK), secure multi-party computation (MPC), or fully homomorphic encryption (FHE) have grown in popularity over recent years. A large class of applications that use these ideas has cryptographic schemes that focus on minimising the number of multiplications. Primarily, there are algorithms that work in a specific field of $\mathbb{F}_p$ or $\mathrm{GF}(p)$ for a prime $p \geq 3$. Working in a Galois field gives a few unique properties since numbers are in a ring structure. Some examples of cryptographic functions which capitalise these properties to lower the number of multiplications include GMiMC[3], Poseidon[2], Griffin[4] and MiMC[1].

The goal to lower the number of multiplications comes from the fact that these algorithms are specifically intended to be optimised for ZK, MPC and FHE. Consider the current block cipher standard AES which with modern hardware equipped with CPU specific instruction sets (AES-NI) can process close to 1GB/s[6] of data. However, in the MPC context, this number is drastically lower which limits the usage of AES in multi-party computation protocols[7]. This comes from the fact that traditionally ciphers are built from linear and non-linear blocks which have similar costs in hardware and software. However, in the MPC or FHE context, linear operations, which can be simply viewed as XOR operations rather than non-linear AND operations, are way cheaper and are

essentially free since they only incur local computation. Thus, the need for ciphers which optimise low multiplicative complexity arises.

1.1 Related work

Secure multi-party computation. The usage of cloud computing over the past decade has grown immensely and it is predicted that by 2028 more than 50% of businesses will use it[24]. One of the cloud computing objectives is for devices to jointly compute some result and be assured that the information is *private* and *correct* which is the goal of secure multi-party computation (MPC) first introduced by Goldreich, Micali and Wigderson[25]. One other usage of MPC, for example, could be to check a person's DNA against a database of cancer patients' DNA, to see if they are in a high risk group for certain cancer types. Clearly, DNA is highly private information and hence should not be revealed, however, using MPC, we can solve this issue by guaranteeing privacy.

MPC is not just a theoretical concept as it has been successfully used in practice. One such example is in Estonia where a privacy preserving study has been conducted using the MPC framework Sharemind[1]. Encrypted income tax together with higher education records have been collected to look for a correlation between working during studies and failing to graduate.[26].

One of the key aspects which is fundamental in MPC is secret sharing which allows multiple parties to share a secret in a way no single party is able to reconstruct the secret on their own. Various secret sharing schemes are used, such as Shamir's Secret Sharing[27]. Another fundamental aspect of MPC is correctness, which provides evidence that neither of the parties cheated or manipulated the data. This is ensured using Zero-Knowledge proofs.

Zero-Knowledge. The term Zero-Knowledge (ZK) proofs were first introduced by Goldwasser, Micali and Rackoff[8] where they give a great example: to prove that a graph is Hamiltonian it suffices to gave a Hamiltonian path, however, this rather gives more information than just the single bit whether the graph is Hamiltonian or not. Thus, loosely speaking, Zero-Knowledge proofs yield nothing more than just the validity of the assertion.

There are various types of proofs available depending on the specific application, each with its own set of characteristics. The selection of a particular type depends on the desired balance between security and performance. For instance, in terms of security, many protocols rely on certain assumptions, often of a computational nature, where solving a mathematical problem is presumed to be difficult. An example of such an assumption is that a hash function behaves like a truly random function[9]. Furthermore, proofs can be categorised as interactive or non-interactive. Interactive proofs involve ongoing communication between a prover and verifier over several rounds, allowing for adjustments to responses based on previous messages. For example, we can interactively prove graph isomorphism[10]. On the other hand, non-interactive Zero-Knowledge proofs (NIZK), introduced by Blum, Feldman, and Micali, are of particular interest to us.

Fully Homomorphic Encryption. One such NIZK is the Boneh-Goh-Nissim[12] encryption which is *somewhat* homomorphic with respect to addition and multiplication respectively. Homomorphic means that computation and analysis can be performed on encrypted data without first decrypting it. There are different levels of homomorphisms from the simplest *partially* homomorphic, *somewhat* homomorphic to *fully* homomorphic encryption (FHE). The latter allows

[1] Privacy focused framework utilising MPC Sharemind.

the computation of data with arbitrary levels of depth or complexity[13]. Note, that the first FHE cryptography system appeared only recently (2009[13]) even though it was first proposed over 45 years ago[14] as privacy homomorphism.

Currently, it is one of the more active research areas in cryptography, as its applications are invaluable to modern day computing. It allows data to be computed in an untrusted environment (such as cloud computing) without breaking confidentiality and preserving privacy concerns. At the moment, the main limitation is the increased computational degree, especially the multiplicative degree, however, works are made to make it more efficient and reduce the overhead when computing with such data[15][16].

SNARKs. There is a problem with non-interactive proof systems when the proof is bigger than the statement itself. To deal with this, we need to make sure that the proofs are brief. If we take NIZK and add another property *succinctness*, then we can also look at a succinct non-interactive argument of knowledge (SNARK) systems. What succinctness means is that the proof is extremely efficient to verify and also much smaller than the statement itself.

One way this property is applied is through the usage of *verifiable computation*, formalised by Gennaro, Gentry and Parno[17]. They describe a system where a computationally weak client can outsource the computationally heavy task to one or more workers, for example in cloud computing environment. Then the workers return the result together with proof that such result is correct. SNARK guarantees that the verification of the proof is easier than doing the task itself. Thus, the client is assured they got the required result without doing heavy computations themselves.

One such almost practical implementation is Pinocchio[18]. Using this system, it is possible to produce a scheme to convince a client that the computation is correct. This is done using a compiler which converts C code into a format suitable for verification. It has been tested on gas simulations, image matching and computing SHA-1 and shown to be faster than native execution showing potential for zero-knowledge applications. Advancements have been to achieve practical implementations of zero knowledge proof systems such as Gepetto[19], MIRAGE[20], Buffet[21] and others. One problem with these systems is that they require a trusted setup, however, there are also *zero-knowledge Scalable Transparent ARguments of Knowledge* (zk-STARK) proof systems, introduced by Eli Ben-Sasso et al[22]. They offer transparency which means anyone can verify the proof without needing any secret information. This is particularly useful in decentralised applications like blockchain. In fact, the first widespread use of zk-SNARK is in the implementation of zerocash blockchain[23].

Bottlenecks. In many scenarios involving MPC, ZK, and FHE, a crucial component of the evaluated circuit is a pseudo-random number function (PRF), a collision-resistant function, or symmetric encryption. Often, the primary bottleneck in these cases is the computational load imposed by multiplications. Thus, research is done to create new ciphers and hash functions which circumvent this weakness. There are encryption schemes such as Ring-LWE[28] that offer lower multiplicative complexity in homomorphic scenarios compared to traditional number theoretic encryption schemes. Then there similar schemes (CKKS[29], BFV[30][31]) and recent improvements[32] which leverage low-degree polynomials. Additionally, there are also the already mentioned cipher schemes Poseidon[2], MiMC[1]. Moreover, cipher families like Rescue together with Vision[33] aim to minimise arithmetic complexity by using simple round function and favouring addition

over multiplication. These families are specifically tailored to be used in zk-STARK and MPC scenarios.

While the theoretical foundations of ZK, MPC, and FHE were laid down in the 1970s and 1980s, practical implementations have only emerged recently. However, as the world transitions more into digital cyberspace, the relevance of these principles is now more important than ever. Privacy and data correctness play a significant role in everyday life. We want to make sure, that our sensitive personal data is handled properly and not leaked to any malicious third party. Therefore, the development of more efficient and optimised encryption schemes is crucial as it allows for widespread tool creation and usage that uphold these concepts.

1.2 Project goals

In this project, we are going to focus on MiMC, which is constructed of the simplified round function of the Knudsen-Nyberg cipher from 1995[5]. The core component of this algorithm is the use of $f(x) = x^3$ APN (almost perfect non-linear) function which has great cryptographic properties[53]. What we are going to do is *build new ciphers based on the structure of MiMC by trying different polynomials to better understand how cryptographic algorithms are proven to be secure and then test cryptographic properties and see whether our newly built cipher is efficient.*

Changing the core function of MiMC which has undergone extensive cryptanalysis with a different polynomial is not enough to say that our new cipher is also secure. First, it is necessary to prove that the function is still a cipher as discussed in chapter 2.2. For this, permutation polynomials of the Galois field are utilised. Then, arguing its security and proving that it holds key cryptographic properties like *diffusion*, *confusion* and *randomness* is required. Arguments for security are given in chapter 3.1, while the key cryptographic properties are tested in chapters 3.2 and 3.4.

To better test the necessary properties, a custom CLI tool is built. The implementation of the tool is given in chapter 4. Using this tool, various tests were run to be more confident whether our cipher is indeed secure. Moreover, the tool allows to compare various polynomials to see which one is the most efficient. It also allows to compare it with the original MiMC cipher together with AES. The tool itself is written in Rust programming language. The choice for this language is the growing popularity due to its high performance, reliability, great community and it is also author's subjective preference. As this project is more theory focused and not coding, the implementation part will discuss some key aspects of implementing core operations and testing them rather than the overall software design.

2 BUILDING NEW CIPHERS

In general, a block cipher can be described as a function of a pair:

$$f : m \times k \to c, \quad f^{-1} : c \times k \to m,$$

where f encrypts the message m with key k and f^{-1} decrypts the ciphertext c. Thus, we have our first definition:

Definition 2.1. A *cipher* is a function f which has an inverse f^{-1} such that $\forall k \in \mathcal{K}$ and $\forall m \in \mathcal{M}$ we have:

$$(f^{-1} \circ f)(m, k) = m.$$

In practice, the domains of $\mathcal{M}$, $\mathcal{K}$ and $\mathcal{C}$ are finite, hence by Universal Mapping Theorem (Theorem 72[34]), any cipher can be written as a polynomial system of equations over $\mathrm{GF}(p)$, for any prime p, where GF is the Galois (finite) field. In general, working in Galois fields is extremely useful, as it allows to construct ciphers and also we will see that it allows for fast computational implementations (as discussed in chapter 4.1). However, first, we need to understand the mathematics of how to work in Galois fields as our cipher will operate in $\mathrm{GF}(p^n)$. We have to properly define such simple things as addition and multiplication once again, as these operations behave differently than in the standard fields of $\mathbb{N}$, $\mathbb{R}$ or $\mathbb{C}$.

After that, we can look at MiMC and its underlying APN x^3. Understanding why this particular polynomial works as a cipher is the key to building new ciphers in the same structure. We need to prove mathematically that for any message m and any key k, we can indeed find the decryption function, such that $f^{-1} \circ f = id$, where id is the identity function. If we do that, then we have successfully created a cipher algorithm. However, this is just the first step of creating a proper cipher algorithm. Then the task is to demonstrate its security which is discussed in chapter 3.

2.1 Field mathematics

As already observed, since the domains of our message $\mathcal{M}$, key $\mathcal{K}$ and ciphertext $\mathcal{C}$ are finite, we will be operating in a Galois (finite) field. However first, it is necessary to understand what a *field* actually is. Thus, an introduction to groups, fields, and the mathematics behind it is given in this chapter. First, we start with groups, hence, a formal definition is given:

Definition 2.2. A group is a set of elements $G = \{a, b, c, ...\}$ and an operation $\oplus$ for which the following axioms hold:

- *Closure*: for any $a \in G$, $b \in G$, the element $a \oplus b$ is in G.
- *Associative law*: for any a, b, $c \in G$, $(a \oplus b) \oplus c = a \oplus (b \oplus c)$.
- *Identity*: There is an identity element 0 in G for which $a \oplus 0 = 0 \oplus a = a$ for all $a \in G$.
- *Inverse*: For each $a \in G$, there is an inverse $(-a)$ such that $a \oplus (-a) = 0$.

At first, it might be confusing what a group actually is but this simply can be the integers under addition, denoted $(\mathbb{Z}, +)$. Note that for any element $a \in \mathbb{Z}$, all the axioms are satisfied as addition does not leave the group, it does not matter in which way we add numbers, 0 is in $\mathbb{Z}$ and we can always find an inverse $(-a)$. However, if we look at natural numbers under addition $(\mathbb{N}, +)$, then this is not a group. This is because the *identity* and *inverse* axioms are not satisfied. For example $5 \in \mathbb{N}$ however, $(-5) \notin \mathbb{N}$, moreover, $0 \notin \mathbb{N}$. Thus, a group consists of two things: a set of elements and an operator which satisfies the four axioms. But we can add more axioms. In particular, if we add the *commutative* axiom we get the *abelian* group:

Definition 2.3. An abelian group G with operator $\oplus$ is a group which additionally satisfies the *commutativity* axiom: for any $a \in G$, $b \in G$, $a \oplus b = b \oplus a$.

This axiom is really useful when working algebraically, as it allows for easier calculations, implementations and will give us fields. Most of the well known groups are abelian $(\mathbb{Z}, +)$, $(\mathbb{R}, +)$, $(\mathbb{Q}, +)$. However, there are groups which are not abelian, namely the non-invertible matrices under multiplication as $AB \neq BA$, for non-invertible matrices A and B.

So far we have talked about groups with infinitely many elements, however, we can have groups with a finite amount of elements $G = \{a_1, a_2, ..., a_n\}$, where $|G| = n$ is the order of G. Moreover, the operator $\oplus$ can then be specified by an $n \times n$ "addition table" whose entry at row i, column j is $a_i \oplus a_j$. The cancellation property implies that if $a_j \neq a_k$, then $a_i \oplus a_j \neq a_i \oplus a_k$. Thus, all elements in any row i of the addition table are distinct, i.e., each row contains each element of G exactly once. Similarly, each column contains each element of G exactly once. Thus, the group axioms restrict the group operation $\oplus$ more than might be immediately evident.

One important example of a finite abelian group is the integers mod-n group denoted $\mathbb{Z}_n$. This is a group where a set is the remainders $\{0, 1, ..., n-1\}$ under mod-n addition, where n is a positive integer. This is actually a finite *cyclic* group, i.e. it has a generator $g \in G$, such that each element of G can be expressed as a sum $g \oplus ... \oplus g$ for some number of repetitions.

Now we can talk about fields and provide a formal definition:

Definition 2.4. A field is a set $\mathbb{F}$ of at least two elements, with two operations $\oplus$ (called addition) and $*$ (called multiplication), for which the following axioms are satisfied:

- The set $\mathbb{F}$ forms an abelian group with identity 0 under the operation $\oplus$.
- The set $\mathbb{F}^* = \mathbb{F} - \{0\} = \{a \in \mathbb{F}, a \neq 0\}$ forms an abelian group with identity called 1 under the operation $*$.
- *Distributive law*: For all a, b, $c \in \mathbb{F}$, $(a \oplus b) * c = (a * c) \oplus (b * c)$.

Hence the field is almost the same as a group but with another operation and a law of how to distribute these two operations. Similarly like with groups, the usual examples for a field are $(\mathbb{R}, +, *)$ and $(\mathbb{Q}, +, *)$. However, $(\mathbb{Z}, +, *)$ does not form a field since there is no multiplicative inverse. This can be fixed if we instead take the cyclic group $\mathbb{Z}_p$ for some prime p. As it turns out, this is indeed a field with operators $+$ and $*$, where multiplication is defined as usual integers multiplication but taking the remainder after division by p.

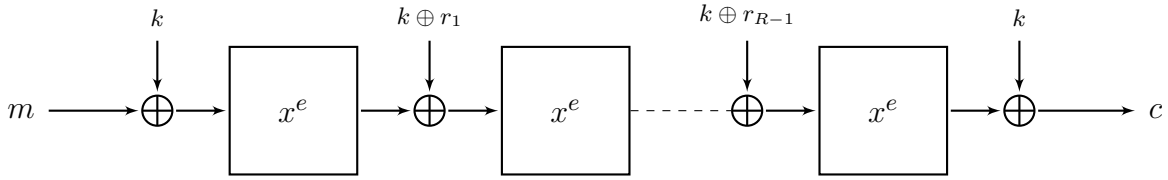
Theorem 2.5. For every prime p, the set $\{0, 1, ..., p-1\}$ forms a Galois (prime) field (denoted by $\mathrm{GF}(p)$ or $\mathbb{F}_p$) under mod-p addition and multiplication.

These fields are actually unique for the given p and can be shown that any field $\mathbb{F}$ with a prime number of elements p is isomorphic to $\mathbb{F}_p$ (by prime field uniqueness theoreom[35]). In fact, this is already enough for us to build a cipher and we can work in $\mathbb{F}_p$ field, however, in this project, we are going to work with $\mathbb{F}_{p^n}$:

Theorem 2.6. If $g(x)$ is a prime polynomial of degree n over a prime field $\mathrm{GF}(p)$ (or $\mathbb{F}_p$), then the set of remainder polynomials with mod-$g(x)$ arithmetic forms a finite field denoted by $\mathrm{GF}(p^n)$ (or $\mathbb{F}_{p^n}$) with p^n elements.

Both of these theorems are proved following the definition, where it is shown, that the field axioms hold for mod-p arithmetic and modular polynomial arithmetic[35]. Some of the proofs are

Fig. 1. MiMC with general exponent e cipher algorithm [1 37].



omitted here and are provided in the references as the goal of the project is not just mathematics. However, modular polynomial arithmetic needs to be discussed next.

2.2 MiMC with general exponent

Modular polynomial arithmetic is done the same way as with integers except we add and divide polynomials. This is because every polynomial $f(x)$ can be expressed as $f(x) = q(x)g(x) + r(x)$ for some polynomial remainder $r(x)$ and polynomial quotient $q(x)$. Then the coefficient operations are simply in $\mathbb{F}$. For example, $x^3 - 2x^2 - 4 = (x^2 + x + 3)(x - 3) + 5$, we can use Euclidean long division to verify this. Now similarly to primes in integers, there are also primes of polynomials.

Definition 2.7. A polynomial is *prime* if it is *monic* and is *irreducible*. That is, the highest order coefficient is 1 and the polynomial has no trivial factors.

Moreover, just like with integers, every monic polynomial can be uniquely expressed as a prime polynomial factorisation. Thus, prime polynomials behave pretty much the same as the usual primes. The only difference is that we are working in a different field. Particularly, for our cipher we are working in $\mathrm{GF}(2^n)$, that is, the message $\mathcal{M}$, key $\mathcal{K}$ and ciphertext $\mathcal{C}$ elements are actually elements of $\mathrm{GF}(2^n)$, for some $n \geq 2$. What this means, is that the elements have the form of $f(x) = a_n x^n + a_{n-1}x^{n-1} + ... + a_0$, where a_0, a_1, ..., $a_n \in \mathrm{GF}(2)$. Any field can be chosen for this, however, implementations of this field are much easier and this is discussed in chapter 4.1. As previously discussed, our operations are modular polynomial operations, thus, addition $+$ and multiplication $*$ is not like with integers.

MiMCGe If we take a look at MiMC cipher[1], it uses the same field $\mathrm{GF}(2^n)$, where n describes the block size. The underlying polynomial for the cipher is x^3. However, we can in fact take any polynomial x^e, for any $e \geq 2$ and we call this the *MiMCGe* cipher. We can prove that this does, in fact, give us a cipher (Fig. 1) by our definition 2.1. The cipher algorithm is simple: plaintext m is added to a random element (round constant) r_i and key k. Then instead of cubic the result, it is raised to some exponent e and this process is repeated for at least $R := \lceil n \log_e(2) \rceil$ number of rounds or for full security: $R + \left\lceil \log_e(\frac{6R}{e}) \right\rceil$. The choice for this is discussed in chapter 3.1. The round constants are chosen as random elements every time we initialise the cipher with the first one being 0. However, these constants could also be predefined earlier similarly to AES.

First, we need to prove that indeed this is a cipher hence let us see why indeed we can take any power e. For this, we need to understand what a permutation polynomial is:

Definition 2.8. A polynomial $f \in \mathrm{GF}(q)$ is called a *permutation polynomial* of $\mathrm{GF}(q)$ if the associated polynomial function $f : c \to f(c)$ is a permutation of $\mathrm{GF}(q)$.

Moreover, remember, that in order to have a cipher, we need to have a function f with an inverse f^{-1}, such that $f^{-1} \circ f = id$. However, if our encryption is essentially a permutation polynomial of the field, then we are guaranteed that there exists an inverse (decryption) function. Luckily, we can use a theorem for this:

Theorem 2.9. A polynomial x^e is a permutation polynomial of the field $\mathrm{GF}(q)$ if and only if $\gcd(e, q-1) = 1$.

The proof of this uses Hermite's criterion[36]. Thus, to have an inverse function, and in turn, have a cipher, it is necessary that the degree e and $q - 1 = 2^n - 1$ are coprime, where n is the block size. Now to build an inverse (decryption) function given $a^e = b$, integer s has to be found such that $b^s = a$, where a and b are in $\mathrm{GF}(2^n)$. However, first additional two theorem are needed:

Theorem 2.10. *(Lagrange's theorem)* If S is a subgroup of a finite group G, then $|S|$ divides $|G|$.

This theorem essentially tells us that the size of any subgroup of a finite group divides the size of the group itself. In other words, if you have a finite group and you consider a subgroup within it, the number of elements in that subgroup must divide evenly into the total number of elements in the original group. Now using this, we can prove the following:

Theorem 2.11. The property $a^{q-1} = 1$ holds for any element $a \in \mathrm{GF}(q)$.

*Proof.*Let the field be $\mathrm{GF}(q)$. Note that since the elements form a cyclic group under multiplication, the equation $x^k = 1$ has at most k solutions in any field, hence, $k \leq |\mathrm{GF}(q)| = q - 1$. Moreover, by Lagrange's theorem 2.10, there exists a divisor k such that $x^k = 1$ for every non-zero x in $\mathrm{GF}(q)$. Putting these two together we get that $q - 1$ is the lowest value to k, i.e., $k = q - 1$. □

Notice, that when q is prime, this is equivalent to Fermat's little theorem and in fact, this is an extension of it to finite fields. Now we can use this to prove our main result:

Theorem 2.12. The inverse element of $b = a^e$ in $\mathrm{GF}(2^n)$ is given by b^s, where $s := \frac{t(2^n-1)+1}{e}$, for some t. If $\gcd(e, 2^n - 1) = 1$, then s and $0 < t < e$ are integers.

*Proof.*It consists of two parts. First, we prove that in fact, s exists, then that s and t are integers. We begin with the fact that we need to find s such that in $\mathrm{GF}(2^n)$, we have:

$$b^s = (a^e)^s = a^{es} = a.$$

Using lemma 2.11, we have that $a^{2^n-1} = 1$ in $\mathrm{GF}(2^n)$. Moreover, we can raise this to some power t and then multiply by a to get:

$$\begin{aligned} a^{2n-1} &= 1 \\ (a^{2n-1})^t &= 1^t = 1 \\ a(a^{2n-1})^t &= a \\ a^{t(2n-1)+1} &= a \end{aligned}$$

Note, that $1^t = 1$ by the second axiom of the field, which says that 1 is the multiplicative identity. Thus we get that $es = t(2^n - 1) + 1 \Rightarrow s := \frac{t(2^n-1)+1}{e}$.

Now to prove that s is an integer we need that:

$$s = \frac{t(2^n-1)+1}{e} \iff t(2^n-1)+1 \equiv 0 \mod e \tag{1}$$

And this can be simplified to $t(2^n-1) \equiv -1 \mod e$. For this, we can use the modular multiplicative inverse property which states that $ab \equiv 1 \mod m$ if and only if gcd(a, m) $= 1$. And this is exactly the same case if we multiply by -1, since gcd(e, $2^n - 1$) $= 1$. Thus, we get that we can indeed find an integer t, such that the equation (1) is satisfied, giving us an integer s. Moreover, if $t \geq e$, then t can be replaced by $t \mod e$ and also, since $\frac{1}{e}$ is not an integer, $t \neq 0$. Giving us that $0 < t < e$. □

This means that since the encryption is a permutation polynomial if and only if gcd(e, $2^n - 1$) $= 1$, then the decryption exponent will also be an integer. Thus, the decryption function is of the same form but with a different power $s := \frac{t(2^n-1)+1}{e}$. Notice, that $e \ll s$, hence decryption is considerably slower and methods to compute fast exponentiation are necessary (algorithm 4 is implemented for this).

3 SECURITY ANALYSIS

What we are now tasked to do is show that the cipher is indeed secure. However, proving security is the hardest step which requires extensive knowledge of cryptanalysis, hence, this project is not a deep dive into the realms of cryptography analysis. Rather, this is just an introduction of what are some techniques that are used to argue and prove that the cipher is secure. Note that, breaking the cipher in cryptography means achieving anything better than brute force attacks in computation or codebook (knowledge of plaintext and ciphertext pairs).

Even the formerly most used ciphers get broken and replaced by other standards. For example DES was broken with the introduction of differential cryptanalysis[38] and later replaced by AES. This is because it is possible to prove that a cipher is secure against some attacks but it is not possible to create a cipher which cannot be broken at all. For that, we need information-theoretic security introduced by Claude Shannon who proved that one-time pad is not breakable with infinite computational power[39]. However, the key needs to be at least as big as the plaintext and can be used only once. Hence, what is more commonly used is the computational security which depends on the fact that some problem is considered ”hard”. For example RSA relies on the fact that large number factorisation is a computationally hard problem. In the case of AES, the confidence of security comes from decades of cryptanalysis with numerous attacks and yet none of them could definitively break it. However, they also could not prove security rigorously. There are some attempts to generalise block ciphers[51] and very recent progress has been made towards provable security with substitution-permutation network (SPN) ciphers[52], but it still remains an open problem. Thus, one of the ways we argue that the cipher is secure is showing that constructing some attack is as computationally hard as brute forcing.

3.1 Round number

Choosing the number of rounds is crucial as we want it to be smaller for efficiency but this comes at the cost of security. Thus, the goal is always to choose it as small as possible to thwart any

possible attacks. A few of these attacks are statistical in nature where they try to predict some behaviour from randomness. These are linear cryptanalysis[40] which tries to approximate cipher as a linear function and differential cryptanalysis[38] which looks at differences through cipher which leads to higher probability in ciphertext allowing to recover parts of the key. However, MiMC was designed based on Knudsen-Nyberg cipher[5] which was specially designed to mitigate differential cryptanalysis. But there are also algebraic attacks such as interpolation[41] or higher-order differential attacks[42][43] which are especially strong against a cryptographic scheme with simple algebraic representation. In fact, Knudsen-Nyberg cipher was broken with the higher-order differential attack[41] and MiMC has also been broken with the same attack[44]. In the paper, it is conjectured that the number of rounds necessary to prevent higher-order differential attacks is also sufficient to prevent interpolation attacks. Thus, the motivation for choosing the round number is based on these attacks.

Higher-Order Differential Attack. The main way this attack works is similar to differential cryptanalysis, however, instead of taking the first derivative, higher-order derivatives are taken. These derivatives are defined as functions f with respect to some $\alpha \in \mathbb{F}_2^n$:

$$\frac{\delta}{\delta\alpha} f(x) = f(x) + f(x+\alpha).$$

Then we can take k-order derivative with elements $\alpha_1, ..., \alpha_k \in \mathcal{V}$ in vector space $\mathcal{V}$ as follows:

$$\frac{\delta}{\delta\alpha_1} \dots \frac{\delta}{\delta\alpha_k} f(x) = \bigoplus_{w \in V + x} f(w).$$

These have useful derivative properties such as the sum rule, product rule and more importantly, taking the derivative reduces the degree. Then, a distinguisher is taken, for example, the zero sum of size s is a set $\{x_1, ..., x_s\} \subseteq \mathbb{F}_2^n$ such that:

$$\sum_{i=1}^{s} x_i = \sum_{i=1}^{s} f(x_i) = 0.$$

If f is truly random, then this should be hard to find, if not, we can distinguish it from randomness. What higher-order differential attack does is exploit the low algebraic degree (definition 3.1) ciphers and build key-recovery attacks based on distinguishers[45]. Thus, it is known that to prevent this attack, the cipher must reach the maximum algebraic degree well within the number of rounds.

Definition 3.1. The *algebraic degree* is the highest degree of any term in the algebraic expression which describes the output bits.

If the structure of the cipher is not trivial, it is difficult to know exactly what this degree is hence, estimations are used. The growth of algebraic degree heavily depends on the cipher structure and a number of rounds. In the original design, it was incorrectly assumed that it grows exponentially in the number of rounds, however, in the MiMC attack[44] it was discovered that it was rather almost linear. In fact, they got that if the number of rounds $r < \lceil \log_e(2^{n-1} - 1) \rceil$, where e is the degree of a round function, then a higher-order distinguisher using at most 2^{n-1} data can be used (Proposition 2[44]). Note, that e always refers to the *exponent* and not the Euler's number. If we set block size to be equal to 129, then this upper bound is $\lceil \log_3(2^{128} - 1) \rceil = 81$, while the

actual number of rounds is $\lceil 129 \log_3(2) \rceil = 82$, leaving only 1 round to break. Breaking the last round is done by building and solving a univariate polynomial $F(K)$ using known-ciphertexts. This requires 2^{n-1} ciphertexts making the attack not practical and it is not intended to replace the original round number estimation as discussed by the authors. However, this gives a great insight of the weakness of the cipher and a new algebraic degree growth bound.

In essence, it means that about $\lceil n \log_e(2) \rceil - 1$ rounds can be set up using zero sum. Thus, the rest of the rounds need to be sufficient enough to make the construction of univariate polynomial $F(K)$ infeasible. We follow a similar way as described in the paper (Section 5.4[44]) but with our general exponent e. We use the fact, that the complexity of constructing the polynomial $F(K)$ is equal to $2^{n-1}(\frac{e^{\rho+1}-1}{2}) = 2^{n-2}(e^{\rho+1} - 1)$, where ρ is the new round number, we want this to be greater than simple exhaustive search $2^n(R + \rho)$, where $R = \lceil n \log_e(2) \rceil$ is the original round number and we get:

$$
\begin{aligned}
2^{n-2}(e^{\rho+1} - 1) &\geq 2^n(R + \rho) \\
e^{\rho+1} &\geq 4(R + \rho) + 1 \\
\rho &\geq \log_e(4(R + \rho) + 1) - \log_e(e) \\
\rho &\geq \log_e\left(\frac{4(R + \rho) + 1}{e}\right) \geq \log_e\left(\frac{6R}{e}\right)
\end{aligned}
$$

Where we suppose that $\rho \leq R/2$. Thus, we get that to thwart higher-order based key-recover attack, it is necessary to take $\rho \approx \log_e(\frac{6R}{e})$ additional rounds. This number is relatively small, for example with block size 129 and exponent 5, $\rho \approx 3$, while with exponent 24 it is only $\rho \approx 1$. Thus, the lower round MiMCGe has $R := \lceil n \log_e(2) \rceil$ rounds, where e is the exponent and n is the block size, while the full round MiMCGe cipher has $R + \lceil \log_e(\frac{6R}{e}) \rceil$ rounds.

3.2 Diffusion and confusion

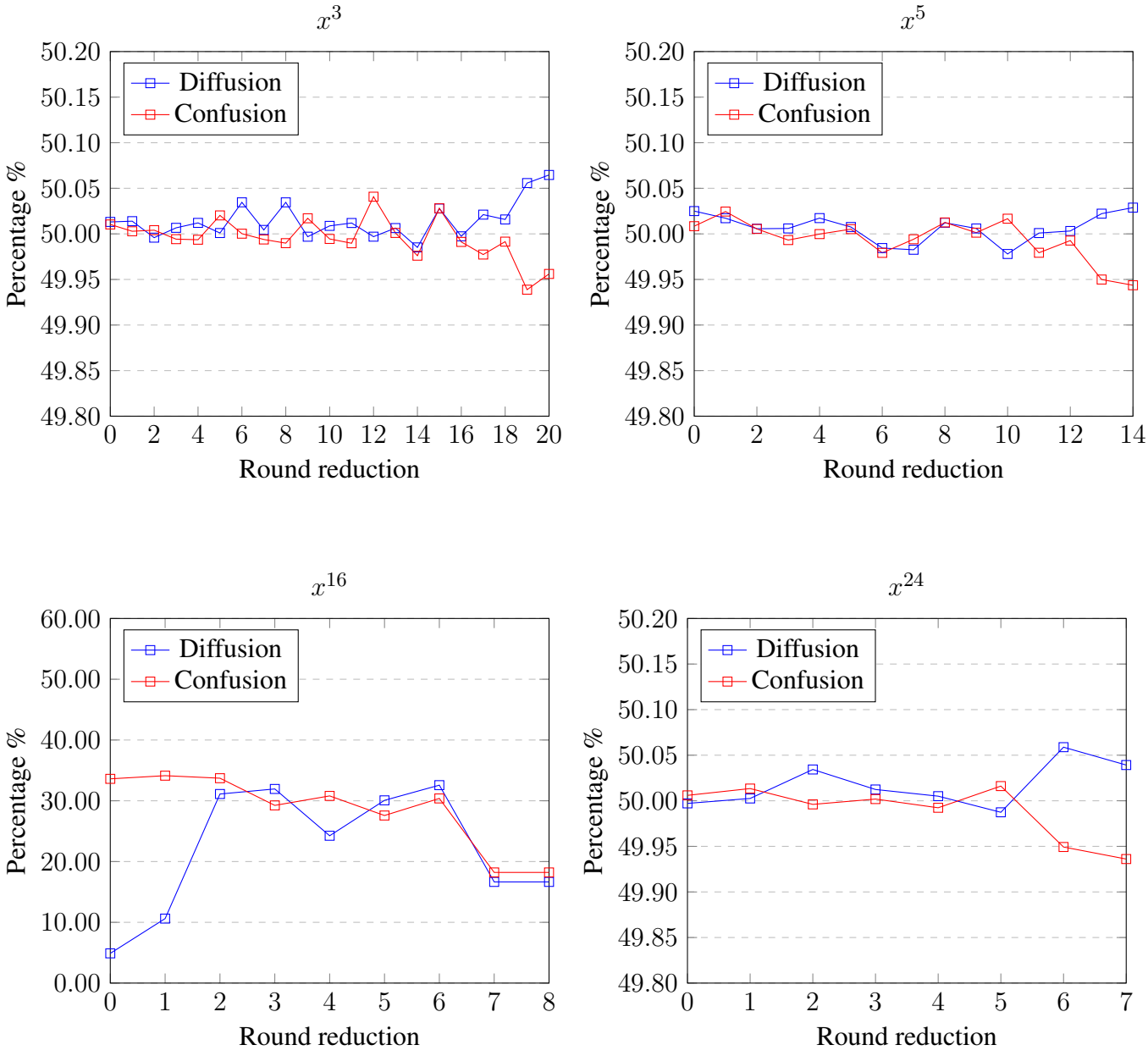
In this chapter instead of mathematically proving cipher security we argue it based on statistical data obtained from testing. The core principle this is based on is Claude Shannon's cipher principle which states that each input and key bit must *influence each* output bit in a *complicated* way[39]. The two properties described here are *diffusion* and *confusion* which can be defined as:

Definition 3.2. *Diffusion* means that a small change in the plaintext results in a significant change in the ciphertext.

Definition 3.3. *Confusion* means that each bit of the ciphertext should depend on several parts of the key, obscuring the connections between the key and the ciphertext.

Diffusion can also be associated with a strict avalanche criterion which states that changing one bit of the plaintext should change each bit of the ciphertext with a probability of one half (a coin flip). Confusion is a similar concept but changing one bit of the key instead of the plaintext should equivalently change about half the bits of the ciphertext. In essence, all secure block ciphers should have these properties. For stream ciphers, diffusion is harder to achieve as they do not work with the whole block. However, since we are working with a block cipher, it is possible to test these properties to see if they hold.

Fig. 2. Diffusion and confusion test results of chosen ciphers



Performing the tests. Two tests were performed: diffusion 1 and confusion 2, where the only difference between them is flipping either plaintext or key bits. The goal is to check how many bits of the new ciphertext change when flipping only a single bit. In the perfect scenario, the goal is to have $r/n^2 = 0.5$, where n is the block size. That is, flipping a single bit is a perfect coin flip of the output. The denominator is squared since the tests iterate over the whole plaintext and flip the bit n times. In reality, it is not possible to check every single plaintext and key pair. This is because there are 2^n unique choices for the plaintext and 2^n unique choices for the key as they are elements of $\mathrm{GF}(2^n)$. It means that there are $2^n * 2^n = 2^{2n}$ different combinations in total assuming round constants are always the same. Hence, only a sample is being tested and the goal is to have **diffusion and confusion rate of** $50\% \pm 0.05\%$.

Algorithm 1 Diffusion test

Require: random plaintext p with key k.

$r \leftarrow 0$
$c \leftarrow encrypt(p, k)$
for i bit in p **do**
  $p_i \leftarrow p[i] \oplus p[i]$ ▷ Flip i^{th} bit
  $c_i \leftarrow encrypt(p_i, k)$
  $r \leftarrow r + \text{count\_flipped\_bits}(c, c_i)$
end for

Algorithm 2 Confusion test

Require: random plaintext p with key k.

$r \leftarrow 0$
$c \leftarrow encrypt(p, k)$
for i bit in k **do**
  $k_i \leftarrow k[i] \oplus k[i]$ ▷ Flip i^{th} bit
  $c_i \leftarrow encrypt(p, k_i)$
  $r \leftarrow r + \text{count\_flipped\_bits}(c, c_i)$
end for

The tests were run $10\,000$ times with four different exponents (x^3, x^5, x^{16}, x^{24}) over the block size 31 for reduced round MiMCGe cipher and the results are displayed in the graphs 2. The exponents were chosen with the intent to compare the original x^3 with others which performed the best as discussed more broadly in chapter 4.3. Additionally, round reductions were applied to see how diffusion and confusion depends on the round number. For example round reduction of 0 means $\lceil 31 \log_e(2) \rceil$ rounds, while 10 for $e = 3$ means $\lceil 31 \log_3(2) \rceil - 10 = 10$ rounds.

From the tests, it is clear that indeed the ciphers have sufficient diffusion and confusion. When no round reduction is applied, all ciphers, except x^{16} which is discussed separately in chapter 3.3, have diffusion and confusion rates of $50\% \pm 0.01\%$ which achieve the predefined goal. Moreover, these diffusion and confusion rates stay consistent as the number of rounds decreases. They do not exceed $50\% \pm 0.05\%$ for almost the whole time. This threshold is exceeded only when the ciphers operate with two or fewer rounds.

It is possible, to verify this rigorously using *null hypothesis* testing. First, denote X_i the diffusion/confusion percentage obtained from test i. Then, we assume that:

$$X_1, ..., X_n \overset{iid}{\sim} N(\mu, \sigma^2)$$

Even if they do not follow the normal distribution, using the central limit theorem and the fact that more than $10\,000$ tests are run, this approximates the normal distribution. Now the null hypothesis is that the mean is fifty percent: $H_0 : \mu = 50$. We pick the level of significance to be $\alpha = 1\%$. Then the null hypothesis is rejected if:

$$|\bar{X} - \mu| > \frac{S}{\sqrt{n}} t_{n-1,\alpha}$$

where $\bar{X}$ is the observed sample mean, S is the sample standard deviation, while $t_{n-1,\alpha}$ is the t-distribution value with $n - 1$ degrees of freedom. The results of the experiments are provided in tables 1 and 2. It is evident, that the result displayed in graphs is not a fluke and the null hypothesis is not rejected with 1% significance level except for the x^{16}.

Function	$\bar{X}$	$\lvert\bar{X}-\mu\rvert$	S	$(S/\sqrt{n})t_{n-1,\alpha}$	Result
x^3	49.9932	0.0068	1.5012	0.0387	Not rejected
x^5	50.0102	0.0102	1.5100	0.0389	Not rejected
x^{16}	48.7603	1.2397	0.0001	0.0000	Rejected
x^{24}	50.0043	0.0043	1.5304	0.0394	Not rejected

Table 1. Results of observed data for confusion.

Function	$\bar{X}$	$\lvert\bar{X}-\mu\rvert$	S	$(S/\sqrt{n})t_{n-1,\alpha}$	Result
x^3	50.0072	0.0072	1.5169	0.0390	Not rejected
x^5	49.9876	0.0124	1.5354	0.0396	Not rejected
x^{16}	39.6694	10.331	0.0000	0.0000	Rejected
x^{24}	50.0151	0.0151	1.5105	0.0389	Not rejected

Table 2. Results of observed data for diffusion.

3.3 Powers of two case

The interesting case is when the exponent is a x^{16}. It is evident, that this is not because the exponent is even, as x^{24} achieves the specified goal of diffusion and confusion. This only happens when exponents are a power of two and it is clear that these ciphers fail diffusion and confusion as they do not even reach 40%, hence we state that **these are not secure ciphers** as they do not show sufficient non linearity. In this chapter, it is analysed rigorously why this happens.

If we look at the definition of the cipher 1, then essentially we can express ciphertext c as:

$$c = \overbrace{f\Big(f\big(...f(m)...\big)\Big)}^{\text{round number of times}} = \Big(...\big((m \oplus k)^e \oplus k \oplus r_1\big)^e ...\Big)^e + k \tag{2}$$

where k is the key, plaintext m and round constants r_i. Then this equation can be expanded using the binomial theorem:

$$(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^{n-k} b^k$$

and then the equation (2) becomes:

$$c = m^{e^r} \oplus \alpha_1 m^{e^r-1} k \oplus \alpha_2 m^{e^r-2} z_1 \oplus ... \oplus \alpha_i z_j \tag{3}$$

where α_k are some constants and z_j is a combination of k and round constants with no m term. So in essence, instead of repeating the rounds, we can just use equation (3), but this is computationally more expensive. However, this can be further simplified using finite fields.

First, using theorem 2.11, every exponent can be reduced by $\mathrm{mod}(2^n - 1)$. This is because $a^{81} = a^{81 \mod 31} = a^{19}$ when operating in $\mathrm{GF}(2^5)$.

Another crucial thing is that these α_k constants can be ignored. The only thing that matters is if they are even or odd. If α_k is even, then it is the same as 0, if it is odd, then it is the same as 1.

This is due to addition in $\mathrm{GF}(2^n)$ which is essentially an XOR operation:

$$(2n)a = (a \oplus a) \oplus ... \oplus a = 0 \oplus ... \oplus 0 = 0$$
$$(2n+1)a = (2n)a \oplus a = 0 \oplus a = a$$

With these two simplifications, we can look at a concrete example. Take exponent $e = 2$, with block size $n = 5$, then equation (3) looks like:

$$c = m^{32} \oplus 32m^{31}k \oplus 16m^{30}k \oplus 496m^{30}k^2 \oplus 4960m^{29}k^3 \oplus$$
$$\oplus 480m^{29}k^2 \oplus ... \oplus k^{16} \oplus 221k^8 \oplus 5k^4 \oplus k^2$$

In total there are over $11\,000$ terms [2] but the main thing to note is that almost all of the coefficients are even which means they get cancelled out. Moreover, the degrees get reduced by $\bmod 31$. In the end, simplifying this yields:

$$c = m \oplus k^{16} \oplus k^8 \oplus k^4 \oplus k^2 \oplus z$$

This is just a linear function of the message. Hence, we get that $c = m \oplus \beta$, where β is some combination of k with r_i.

If we compare this with exponent $e = 3$ and the same block size $n = 5$, then (3) is:

$$c = m^{81} \oplus 81m^{80}k \oplus 3240m^{79}k^2 \oplus 27m^{78}k \oplus 2106^{77}k^2 \oplus$$
$$\oplus 351m^{75}k^2 \oplus ... \oplus 3m^{27}k^2 \oplus 27m^{24}k^3 \oplus 9m^{18}k^3 \oplus k^3$$

The thing to note is that simplifying this is much much harder because there are terms with odd coefficients. This means, that the ciphertext does not depend on just a single entry m like before and thus, the resulting equation is not linear.

The question then is, why all terms are even, when the exponent is a power of two? To answer, we use a special case of Lucas' theorem[56]:

Theorem 3.4. Let n and k be non-negative integers. Then

$$\binom{n}{k} \equiv \begin{cases} 0 \mod 2 & \text{if } n \text{ is even and } k \text{ is odd} \\ \binom{\lfloor n/2 \rfloor}{\lfloor k/2 \rfloor} \mod 2 & \text{otherwise} \end{cases}$$

The key point is that when the exponent is a power of two, then from the theorem it follows that n is always even and has greater power of two than k, hence $\binom{n}{k}$ will always be even. This in turn means, that equation (3) will always have even coefficients except when the term does not have m. In the end, for all exponents of powers of two, the resulting equation will have only one m term. This means, that c is a simple function of m:

$$c = m^\gamma \oplus \beta$$

for some exponent $\gamma < 2^n - 1$ and constant β.

This explains why diffusion and confusion rates are the same for every test size. As there is only one m term, flipping one bit does not propagate throughout the whole circuit. It also explains

[2] This was checked using WolframAlpha.

why confusion rates are better as key bits have more influence. This also shows why increasing the round number increases security. The larger the round number, the more terms there are in equation (3). Moreover, creating an attack against this cipher is relatively easy. As discussed before, the maximum algebraic degree (definition 3.1) must be reached, but in this case it is low. Therefore, exponents of a power of two are not secure ciphers.

3.4 Pseudo-random number generator

Another way the ciphers can be looked at is through random number generators. If a cipher is truly secure, then it must be non-linear and the mapping of the input must be random. That is, if we give an input, it should map to the output in a random and unpredictable way. Hence, a cipher is also a pseudo-random number generator. This is, however, a consequence of producing a cipher rather than the actual objective. Thus, when arguing security, formal mathematical analysis is always more prioritised over statistical analysis. However, it is a great tool which can easily demonstrate if the cipher is insecure.

Note, that it is impossible to generate truly random numbers in any mathematical way. In fact, even in physics, the sources of true randomness are rare. Hence, the term *pseudo*-random number generators (PRNG) is used when speaking about generators, which are software based. The study of randomness is an active field of research with recent papers using quantum physics for zero-knowledge proofs[46] or developing generators for FHE[47]. Thus, there are tools created specifically for testing these generators, notably NIST STS[48] and Dieharder[49] which are considered to be industry standard for testing cryptographic algorithms[6].

The underlying theory which all of these testers utilise is the *Central Limit Theorem* (CLT). The idea is that the PRNG produces a stream of some numbers on a specific range (floating point numbers, 32 bit, etc.). Then to construct an experiment, sum up all the numbers to get t which should give a mean value of $\mu = t/2$. Now taking a large amount of these experiments produces independent and identically distributed sums t_i. By the CLT, if PRNG indeed produces uniformly distributed numbers, the standard deviation is $\sigma = \sqrt{t/12}$. That gives the *null hypothesis* such that:

$$t_1, ..., t_n \overset{iid}{\sim} N(\mu, \sigma^2)$$

Now a p-value test is constructed which gives the probability of obtaining the sum x from some experiment t_i. This is given by:

$$p = \operatorname{erfc}\left(\frac{|\mu - x|}{\sigma\sqrt{2}}\right)$$

If this value is really low or really close to 1 with predetermined level of significance α, then it means that the PRNG is biased towards some particular numbers, thus, the null hypothesis is rejected. However, passing the test does not mean that the PRNG is indeed random. With null hypothesis testing, only rejecting is possible. If the null hypothesis is not rejected, it only gives higher certainty, that it is indeed true. Thus, numerous tests are performed to increase this confidence.

NIST STS. It is a collection of 15 tests developed by the National Institute of Standards and Technology (NIST) aimed at assessing the randomness or statistical properties of PRNG or cryptographic algorithms. The tests require an input file at least of $10\,000$ random stream of

Function	Pass	Fail	Result	Rate
x^3	15	0	187/187	100%
x^5	15	0	186/187	99.47%
x^{16}	1	12	1/162	0.62%
x^{24}	15	0	187/187	100%

Table 3. NIST test results.

Function	Pass	Weak	Fail
x^3	111	3	0
x^5	111	3	0
x^{16}	3	5	106
x^{24}	111	3	0

Table 4. Dieharder test results.

bits. For accurate results, there have to be at least 55 bit streams (as stated in NIST manual Section 4.2.2[48]). In MiMCGe case, four different exponents (x^3, x^5, x^{16}, x^{24}) were tested with a block size of 127. 1000 bit streams were generated of 7875 different numbers which yield to $127 * 7875 = 1\,000\,125$ size per bit stream. In total, the input size is ∼1GB. The generated file simply consists of encrypted sequential numbers, where the random seed corresponds to the key together with round constants. This seed was chosen randomly and is the same for all the different exponents and tests. The implementation of generating this file is discussed in chapter 4.4.

At first, two control tests were performed. First is the built in test which checks a known PRNG against the test suite to check whether the tests correctly identify PRNG. The second control test was to see whether the construction of the input file is generated correctly. For this, the numbers were not encrypted and a file of just sequential numbers was given to the the test suite. This correctly identified that 15 out of 15 tests fail. Now, the files with encrypted numbers were given and the results are provided in table 3. The result column indicates how many of the test experiments were passed, out of all performed. Note, that some tests have multiple experiments (i.e. *Non-overlapping Template Matching Test* has 147 experiments). The only experiment x^5 failed was the non-overlapping template experiment, however, even then, the result was 979/1000, while the pass rate is 980/1000[3]. It is evident, that indeed, as previously discussed, powers of two exponents are not secure ciphers. The test suite could not even perform the *Random Excursions* and *Random Excursions Variant* tests for x^{16}. Interestingly, x^{16} passed the *Binary Matrix Rank Test* for NIST STS, however, a similar test is in the dieharder battery test and it failed it there. This is likely due to insufficient input data discussed next. However, NIST STS test suite gives more confidence that the other ciphers are indeed secure.

Dieharder. It is another battery of statistical tests for pseudo-random number generators. Dieharder was developed as an extension of the original Diehard battery of tests by George Marsaglia[50]. This test suite is similar to NIST STS, however, it has 31 different tests and the biggest difference is the input size. This test suite requires considerably larger sample space which can be anywhere close to ∼240GB. If the input file is smaller, then the tests rewind the file and go through it again. This can cause inaccurate results, however, the test suite also works with a random bit stream. Hence, for this test, instead of a file, a stream of encrypted sequential 32 bit numbers was provided. Implementation details are discussed in chapter 4.4.

Once again, the same two control tests were performed to check the test suite and the implementation of the input. Then the same exponents (x^3, x^5, x^{16}, x^{24}) were tested and the results are given in table 4. Note that, since these tests require considerably larger input, it took

[3] Full report of the test suites is available on GitLab

over a week to get the full results, while NIST STS took around 8 hours. From the results, it is evident, that the ciphers behave like a PRNG. This battery of tests additionally says whether the test was passed with low confidence (weak column). This simply means, that the p-value has higher level of significance α. But it still means, that the test has been passed, just with lower confidence. However, as before, it is evident, that once again we are proven that x^{16} is not a secure cipher, while other exponents perform as expected.

In general, these two test suites give a way to verify that a cipher does behave like a pseudo-random number generator. However, it is important to recognise, that they do not inherently prove security. The underlying test which they perform is the hypothesis testing, which can only reject the hypothesis. Therefore, it is impossible to prove security using these tools. But even so, when these tools fail to disprove the hypothesis, they in turn give more confidence, that the ciphers are in fact secure.

4 IMPLEMENTATION

After completing the theoretical groundwork, it is necessary to move to the practical cipher implementation. This comes with its own challenges when tackling software design and optimisation problems. In this project, a custom Command Line Interface (CLI) tool was built which implements MiMCGe cipher[4]. It allows to test encryption and decryption, run diffusion and confusion tests, generate test inputs for the NIST STS and dieharder test suites. For comparison, AES and original MiMC ciphers are also included which can be used in place of MiMCGe in all the tests above.

4.1 Design

The git project[4] has five directories:

- */scripts*. Contains the utility shell scripts which were run to get test results for diffusion and confusion and to generate test data for statistical tests.
- */src*. Contains the implementation of the CLI tool of the cipher written in Rust.
- */test-suite-results*. Contains the results generated from the statistical test suites.
- */tests*. Contains integration tests for the CLI tool.
- */report*. Contains this report in LaTeX and in PDF format.

The CLI tool is written in Rust, thus, for the user to compile the code, *rustup*[5] needs to be installed first. There is a supplementary *README.md* file which has a user manual together with some useful examples. Moreover, the code is well documented and the tool itself provides help when run with the `--help` flag. For example, encrypting 2026 with key 432, exponent x^7 in a block size of 11 can be run as:

```
$ ./mimcge cipher-test mimcge 11 -p 2026 -e 7 -k 432
```

[4] The code is accessible on Gitlab.

[5] Toolchain which runs and compiles Rust code.

```
Plaintext:  2026 [1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0]
Ciphertext: 1067 [1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1]
Decrypted:  2026 [1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0]
Time: 20.67µs
```

The program displays the plaintext both in decimal form and as a bit vector. It then presents the corresponding ciphertext and its bit vector, along with the decrypted ciphertext. Additionally, the program reports the time it took for encryption and decryption processes.

Implementing field arithmetic. As discussed in chapter 2.1, basic operations are not the same as with standard fields $\mathbb{N}$ or $\mathbb{R}$. This was the initial hurdle when trying to implement the cipher. Thus, to properly build the cipher, field arithmetic needs to be implemented from scratch. We start by creating a type for the element of the field: `FieldElement` which will essentially be a vector of 1s and 0s (bits). This is because the elements of the Galois field $\mathrm{GF}(2^n)$ are polynomials, thus, the elements in the bit array represent the coefficients of the polynomial. That is, if we have an element $x^5 + x^3 + x^2 + 1 \in \mathrm{GF}(2^3)$, then the corresponding `FieldElement` will be of length $2^3 = 8$ and look as: `[0, 0, 0, 1, 0, 1, 1, 0, 1]`. This shows one of the benefits of implementation when working with the finite in the form of $\mathrm{GF}(2^n)$, as elements are just simple bit arrays.

Now addition, multiplication and exponentiation need to be implemented as these are the only operations the cipher requires. First, notice, that adding two polynomials is essentially a XOR operation of the coefficients. For example, adding $x^5 + x^3 + x^2 + 1$ with $x^4 + x^2$:

$$\begin{array}{rl} & 1x^5 + 0x^4 + 1x^3 + 1x^2 + 0x + 1 \\ + & 0x^5 + 1x^4 + 0x^3 + 1x^2 + 0x + 0 \\ \hline & 1x^5 + 1x^4 + 1x^3 + 0x^2 + 0x + 1 \end{array}$$

Thus, implementing this is rather simple and this is yet another benefit of using the powers of two for the finite fields.

However, with multiplication, problems arise. This is because when two polynomials are multiplied, the total degree increases, however, to not break the closure axiom of the field, elements must stay in the field. To tackle this problem, the resulting polynomial needs to be reduced by an irreducible polynomial (definition 2.7). This is similar to modular arithmetic, where $a * b$ is defined as $(a * b) \mod p$, for some prime p. There are a few ways to implement this, including the naive implementation following the definition, however, the more efficient one is the one using the peasant algorithm 3. This algorithm utilises doubling and halving numbers while discarding the remainder, which is very efficient since it is essentially only shifting bits left or right. To apply this algorithm for finite field arithmetic, the only two changes needed is changing the addition, which, as discussed previously, is the XOR operation. Another change needed is to check if a escapes the field when doubling. That is, if working in $\mathrm{GF}(2^3)$ and a already has the term x^7, then after doubling, it must be reduced by an irreducible polynomial. Note that, doubling a is the same as with integers: shifting bit to the left. It also means, that for every different finite field, an irreducible polynomial must be given. There are ways to generate such polynomials, however, they were supplied beforehand. This means that only selected $\mathrm{GF}(2^n)$ are implemented and since

n means the block size, only some number of block sizes are valid as multiplication is not defined for others.

The only operation left is exponentiation. This can also be implemented naively using the definition $a^e = \overbrace{a * ... * a}^{e}$ since multiplication is already defined. However, notice, that the decryption exponent is considerably larger (i.e. for block size $n = 33$ and exponent $e = 5$, decryption exponent $s := \frac{t(2^n-1)+1}{e} = \frac{4(2^{33}-1)+1}{5} = 6871947673$). It was tested that with naive implementation, decryption of block size 33 took $\sim$10.5 hours. Thus, some method to minimise the number of multiplications is needed. There are a few ways to do this. One of the more popular methods is using pre-computed tables. However, they require a lot of data and also since this tool allows variable block size, the same peasant algorithm was chosen but with multiplication instead of addition. This is actually the square and multiply algorithm 4. This algorithm similarly to peasant algorithm, utilises halving and doubling. One other advantage is that since multiplication takes care of the field axioms, no modification for finite field arithmetic is needed. This algorithm brings down multiplications to around $\mathcal{O}(\log(e))$, instead of linear $\mathcal{O}(e)$, where e is the exponent. This allows to bring back decryption into practical use and the same decryption of block size 3 now takes $\sim 200\mu s$.

Algorithm 3 Peasant multiplication

Require: a, b.
  $result \leftarrow 0$
  while $a > 0$ and $b > 0$ **do**
    if b is odd **then**
      $result \leftarrow result + a$
    end if
    Double a
    Divide b by half discarding the remainder
  end while

Algorithm 4 Square and multiply

Require: a, $exponent$.
  $result \leftarrow 1$
  while $exponent > 0$ **do**
    if $exponent$ is odd **then**
      $result \leftarrow result * a$
    end if
    Double a
    Divide $exponent$ by half discarding the remainder
  end while

One important thing to note is that these algorithms do not operate in constant time. This can lead to side-channel attacks which exploit not the cryptographic algorithm itself but the implementation of it. This is a fascinating topic with lots of active research which exploit microchip processors to steal AES and RSA keys[54] and the most recent vulnerability found in the Apple silicon chips[55]. However, in our algorithms, because of the if statement in the loop, this implementation could be exploited with a timing attack. That is, the attacker by measuring the time taken for certain operations could infer information about the key or the input data. To mitigate these issues there are different variations of these algorithms which are not discussed here as it deviates from the goal of the project.

4.2 Testing

A critical aspect of software development is testing, which typically involves two main types: *unit* testing and *integration* testing. Unit testing focuses on just tiny portions of the code, for example, if converting a binary array to the decimal expression is implemented correctly. On the other

hand, integration testing examines from the user's perspective, i.e. running the tool produces the expected output. Every good software should have both types of tests. How much testing is done, depends on the programmer as too many can slow down development, while not enough run the risk of having software with bugs. Thus, only the key parts of the software are thoroughly tested.

The integration test file, following the conventional Rust file structure, is located in the */tests* directory. It contains simple tests which check that then the tool is run with specific block size, key, plaintext and round constants, and the expected value is printed to the standard output. This is the *happy path* testing, where with the predefined inputs, the program successfully finishes, providing the expected output. Additionally, tests are run which intentionally expect the program to fail. In that case, an error message is checked whether it has an expected output. This way the CLI tool is checked whether it does not break and the proper error message is displayed for the user.

The more interesting tests are the unit tests. These are located in the */src/tests* directory. There is a larger amount of these tests than integration tests. This is because every aspect of field arithmetic is checked. These tests check that the five main axioms (associativity, identity, inverse, commutativity, distributivity) hold for every implemented block size. These axioms are checked for both addition and multiplication as these are the two main operations needed. If all of these axioms hold for the block size, then it is guaranteed, that the provided irreducible polynomial is correct for the specific block size and finite field arithmetic is implemented correctly. For this reason, every block size must be tested separately. Additionally, various helper functions, such as converting an array of bits to decimal expression and vice versa are tested as these two functions are used intensely throughout the program.

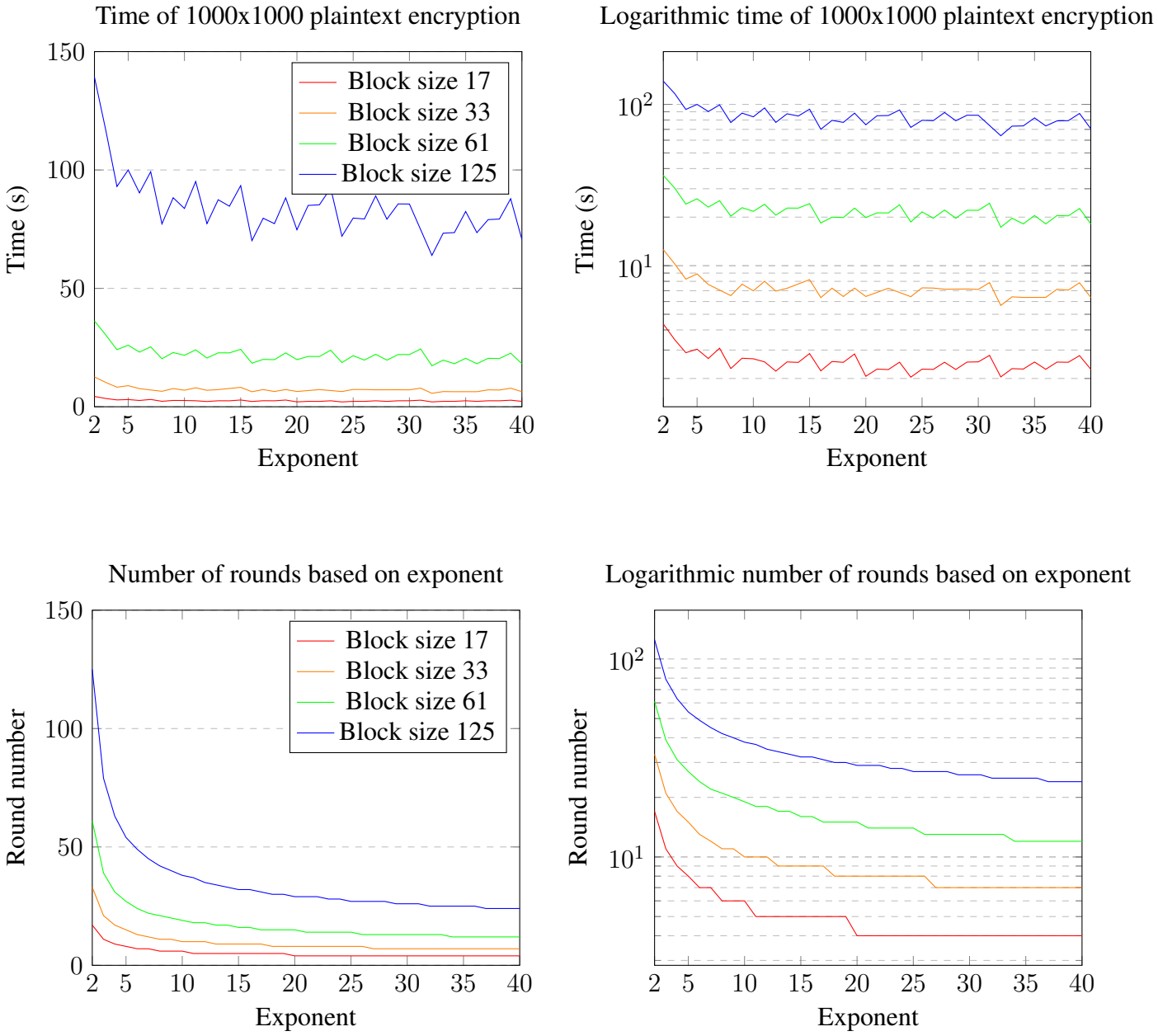
4.3 Choosing the exponents

After implementing and testing that it was done correctly, it is time to compare the ciphers of different exponents to choose the most interesting ones for further analysis. First, the time it takes for the ciphers to encrypt some amount of plaintexts was compared based on the exponent. Four different block sizes were chosen (17, 33, 61 and 125). Then a test is: build a cipher with an exponent ranging from 2 to 40 with random key and round constants. Then count the amount of time it takes for the cipher to encrypt $1\,000$ random plaintexts. This test is repeated $1\,000$ times for each cipher, yielding that each function x^e encrypts $1000 * 1000 = 1\,000\,000$ plaintexts. The results of this are given in the graphs 3.

The top graphs show the encryption test results, while the bottom graphs are the round number functions $R = \lceil n \log_e(2) \rceil$, where e is the exponent and n is the block size. The graphs on the right are just the graphs on the left with a logarithmic scale for better visualisation. These graphs showcase a few interesting things. First, it is evident, that as the exponent increases, the round number decreases and hence the amount of time it takes for the test also decreases. This is pretty obvious, as the number of rounds is lower, there are fewer computations. However, looking at the logarithmic scale, it is evident, that as the exponent is increased further, the round number decreases, but the time it takes to encrypt does not decrease much further. Hence, exponents > 30 do not bring further optimisations and we chose x^{24} for further analysis.

Another interesting thing to note is that there are these dips in time which are evident in the top left graph. These sudden dips are for x^4, x^8 and all x^{2^a} (the powers of two). This is, however, a consequence of the implementation. The cause of this is the square and multiply algorithm 4.

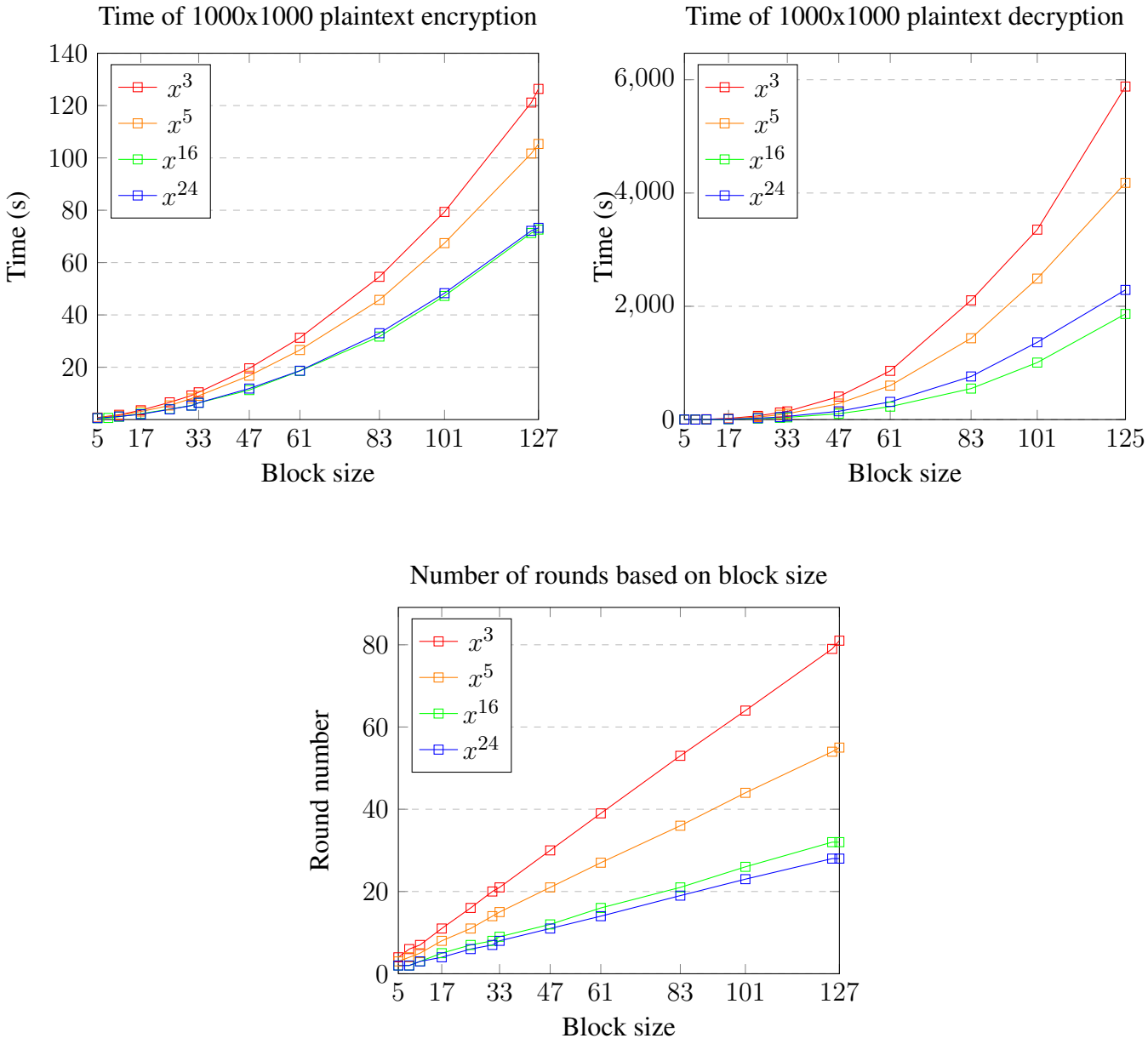
Fig. 3. Encryption and number of rounds based on exponent (tested with AMD Ryzen 5 4500U CPU).



This algorithm reaches its best case when exponents are the powers of two because then there are zero multiplications and the number is only doubled. Thus, we choose x^{16} for further analysis.

Another test we can perform is when the dependencies are flipped. That is, encryption time dependency on the block size. In this case, we choose x^3 and x^5 to check how they differ, together with previously chosen x^{16} and x^{24}. Then the test is the same, where $1\,000$ different ciphers are built with random key and round constants (essentially a random seed) and the time it takes for them to encrypt $1\,000$ random plaintexts is counted. Each implemented block size (5, 8, 11, 17, 25, 31, 33, 47, 61, 83, 101, 125 and 127) is tested and the results of this is provided in graph 4. Additionally, decryption time is compared. This gives a great visualisation, that even though number of rounds is a linear function, the encryption and decryption time grows exponentially.

Fig. 4. Encryption/decryption and number of rounds based on block size (tested with AMD Ryzen 5 4500U CPU).



Another interesting detail is that as the exponent increases, the encryption and decryption time decreases, however, x^{16} is slightly faster than x^{24}. This is, as stated before, because of the square and multiply algorithm 4 efficiency. Moreover, it is evident, that decryption is much slower and this is because the algorithm is essentially the same where only the exponent is different and it is much larger.

4.4 Generating test data

As discussed in chapter 3.4 about statistical test suites, a sample space needs to be provided. Here it is explained more broadly how these inputs were generated.

With NIST STS, generating the input files is simple. A utility CLI command `generate-test-samples` was created which simply loops through the numbers from 0 up

to some specified t. Then encrypts them with the given cipher and prints the result to the standard output. For this, a shell script *generate-numbers-list.sh* was created which calls the CLI with the predefined cipher and its parameters and pipes the result to a file. The parameters are the 127 bit key and round constants which represent a random seed. For every different function x^e, the same seed was provided. The only difference was the number of round constants, as these represent the number of rounds.

Providing input for dieharder statistical test suite is a bit more complicated. Initially, the same setup was used, however, then the test files were too small and were rewound up to 7800 times which caused inaccurate results. For this reason, another method to provide the sample space was needed. Besides the input file, an endless bit stream can also be provided to the dieharder test suite. This stream must be of 32 bit numbers generated by the pseudo-random number generator being tested. For this, `start-bit-stream` CLI command was created which does exactly that. The issue is, however, that some exponents do not work with 32 bit block size (i.e. x^3) because the function is not a permutation polynomial. To circumvent this, 33 bit block size is used and then one bit is dropped when outputting the result to the standard output. Another potential issue is that as the test takes a long time to finish, it is possible that when iterating over the sequence of bits, the iterator of the sequence might overflow. To prevent the sequence iterator from overflowing, it is reduced by a modulo $p = 4\,531\,145\,293$. This number is chosen specially to be a prime larger than 2^{32} and less than 2^{33} which means that the iterator forms a ring $\bmod p$. In this case, the numbers do eventually start to repeat. However, the statistical test suite uses less than 2^{32} inputs, thus, this does not affect the test results. Similarly as before, shell scripts were created which contain the random seed. Moreover, another shell script was created to run some of the experiments in parallel as some of them took more than 8 hours.

5 CONCLUSION

Recall, that the main goal of the project was: to build new ciphers, argue their security and test cryptographic properties (chapter 1.2). And in fact, this goal was achieved. It was not an easy task, as it required to learn cryptanalysis with the guidance of supervisor. Another help was the cryptanalysis course[45] by the same researchers who in part created and found an attack on MiMC. However, the journey began by first proving that in the chosen field $\mathrm{GF}(2^n)$ it is possible to find any function which will be a cipher by definition 2.1. For that, permutation polynomials of any exponent were chosen and this gave the *MiMCGe* cipher (chapter 2.2). Then the task was to prove that these class of ciphers are indeed secure.

However, as discussed in chapter 3, definitively proving security for symmetric key ciphers is an open problem. Information-theoretic security has been proved only with the one-time pad[39]. Recent progress has been made with substitution-permutation network ciphers but to the best of our knowledge, there is no definitive way of proving security. Because of this, arguing that the cipher is secure against some well known attack paradigms is the best cryptanalysis tool. Additionally, as a consequence, main cryptographic properties, such as diffusion, confusion and randomness can be tested.

The main argument of security is in chapter 3.1 where the number of rounds for the cipher was chosen. Then, it is proved that this round number is sufficient to thwart algebraic attacks, especially higher-order differential attack. The proof is based on the original MiMC paper [1] together with an attack found of the original cipher[44] where ciphers with simple algebraic representation are found to be weak against algebraic attacks.

Having decided full specification of the cipher, the main cryptographic properties were tested. As discussed earlier, the core principle of the cipher is that each input and key bit must influence each output bit in a complicated way. These are the diffusion and confusion properties which were analysed in chapter 3.2. The tests were constructed and it was proven that sufficient diffusion and confusion rates are reached.

Another property tested of the ciphers was that they behave similarly to a pseudo-random number generator. Two industry standard statistical test suites were utilised for this task. It was shown, that indeed the ciphers behave like a PRNG which means that they behave like a non-linear function. However, one important thing to note is that these tools do not prove security, they can only disprove it. Null hypothesis testing is similar to proving by contradiction. First, you assume a statement and then derive a contradiction, proving the statement is false. However, if you cannot get a contradiction, it does not mean that the assumption is correct.

Performing these tests, it was shown, that when the exponent is a power of two (x^2, x^4, etc.), the ciphers are not secure. It was further analysed and rigorously proved why that is the case. This not only demonstrates practical cryptanalysis but also showcases how ciphers operate internally and how difficult it is to argue security. It demonstrates why using various tools is necessary as it helps to quickly spot vulnerability.

Having analysed security, the ciphers were tested for efficiency. A CLI tool was written in Rust which allowed to construct tests of the cipher. The relation between the exponent and block size was compared. The results show that as the exponent increases, the time it takes for the cipher to encrypt decreases. Additionally, smarter algorithms had to be utilised in order to make decryption even possible. Some further optimisations can be used with pre-computed multiplication tables, hardware accelerated multiplication or other algorithms for field arithmetic. However, making the most optimised implementation was not the objective of the project.

In the end, the main aim of these types of ciphers is not the quickest way to encrypt a million bits. These ciphers are implemented with the intent of utilising them for zero-knowledge, multi-party computation or fully homomorphic encryption. As discussed in the beginning, modern cryptography research focuses beyond the classic confidentiality and authenticity between two-party communication problems. Considerable effort is made to find better algorithms for ZK, MPC, SNARK, etc. which in turn provide more security and privacy for everyday internet users and beyond. This project is just a small glimpse into the realms of cryptanalysis and ways of constructing cryptographic algorithms.

REFERENCES

[1] Martin R. Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, Tyge Tiessen. *MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative*

Complexity, URL, 2016.

[2] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, Markus Schofnegger. *POSEIDON: A New Hash Function for Zero-Knowledge Proof Systems*, URL, August 2021

[3] Martin R. Albrecht, Lorenzo Grassi, Léo Perrin, Sebastian Ramacher, Christian Rechberger, Dragos Rotaru, Arnab Roy, Markus Schofnegger. *Feistel Structures for MPC, and More*, URL, 2019.

[4] Lorenzo Grassi, Yonglin Hao, Christian Rechberger, Markus Schofnegger, Roman Walch, Qingju Wang. *Horst Meets Fluid-SPN: Griffin for Zero-Knowledge Applications*, URL, 2022.

[5] Kaisa Nyberg, Lars Ramkilde Knudsen. *Provable security against a differential attack*, URL, December 1995.

[6] Eslam G. AbdAllah, Yu Rang Kuang, Changcheng Huang. *Advanced Encryption Standard New Instructions (AES-NI) Analysis: Security, Performance, and Power Consumption*, URL, February 2020.

[7] Benny Pinkas, Thomas Schneider, Nigel P. Smart, Stephen C. Williams. *Secure Two-Party Computation is Practical*, URL, 2009.

[8] Shafi Goldwasser, Silvio Micali, Charles Rackoff. *The Knowledge Complexity of Interactive Proof Systems*, URL, February 1989.

[9] Mihir Bellare, Moti Yung. *Certifying Permutations: Non-Interactive Zero-Knowledge Based on any Trapdoor Permutation*, URL, 1992.

[10] Jonathan Bootle, Andrea Cerulli, Pyros Chaidos, Jens Groth. *Efficient Zero-Knowledge Proof Systems*, URL, August 2016.

[11] Manuel Blum, Paul Feldman, Silvio Micali. *Non-interactive zero-knowledge and its applications*, URL, January 1988.

[12] Dan Boneh, Eu-Jin Goh, Kobbi Nissim. *Evaluating 2-DNF Formulas on Ciphertexts*, URL, April 2006.

[13] Craig Gentry. *Fully Homomorphic Encryption Using Ideal Lattices*, URL, May 2009.

[14] Ronald L. Rivest, Len Adleman, Michael L. Dertouzos. *On data banks and privacy homomorphisms*, URL, October 1978.

[15] Craig Gentry, Amit Sahai, Brent Waters. *Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based*, URL, June 2013.

[16] Jung Hee Cheon, Andrey Kim, Miran Kim, Yongsoo Song. *Homomorphic Encryption for Arithmetic of Approximate Numbers*, URL, November 2017.

[17] Rosario Gennaro, Craig Gentry, Bryan Parno. *Non-Interactive Verifiable Computing: Outsourcing Computation to Untrusted Workers*, URL, February 2010.

[18] Bryan Parno, Jon Howell, Craig Gentry, Mariana Raykova. *Pinocchio: Nearly Practical Verifiable Computation*, URL, May 2013.

[19] Craig Costellom, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, Samee Zahur. *Geppetto: Versatile Verifiable Computation*, URL, May 2015.

[20] Ahmed Kosba, Dimitrios Papadopoulos, Charalampos Papamanthou, Dawn Song. *MIRAGE: Succinct Arguments for Randomized Algorithms with Applications to Universal*

zk-SNARKs, URL, March 2020.

[21] Riad S. Wahby, Srinath Setty, Zuocheng Ren, Andrew J. Blumberg, Michael Walfish. *Efficient RAM and Control Flow in Verifiable Outsourced Computation*, URL, February 2015.

[22] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, Michael Riabzev. *Scalable, transparent, and post-quantum secure computational integrity*, URL, March 2018.

[23] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, Madar Virza. *Zerocash: Decentralized Anonymous Payments from Bitcoin*, URL, November 2014.

[24] Gartner. *Gartner Says Cloud Will Become a Business Necessity by 2028*, URL, November 2023.

[25] Oded Goldreich, Silvio Micali, Avi Wigderson. *HOW TO PLAY ANY MENTAL GAME*, URL, January 1987.

[26] Dan Bogdanov, Liina Kamm, Baldur Kubo, Reimo Rebane, Ville Sokk, Riivo Talviste. *Students and Taxes: a Privacy-Preserving Social Study Using Secure Computation*, URL, December 2015.

[27] Adi Shamir. *How to share a secret*, URL, November 1979.

[28] Vadim Lyubashevsky, Chris Peikert, Oded Regev. *On Ideal Lattices and Learning with Errors Over Rings*, URL, June 2013.

[29] Jung Hee Cheon, Andrey Kim, Miran Kim, Yongsoo Song. *Homomorphic Encryption for Arithmetic of Approximate Numbers*, URL, May 2016.

[30] Zvika Brakerski. *Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP*, URL, February 2012.

[31] Junfeng Fan, Frederik Vercauteren. *Somewhat Practical Fully Homomorphic Encryption*, URL, March 2012.

[32] Beatrice Biasioli, Chiara Marcolla, Marco Calderini, Johannes Mono. *Improving and Automating BFV Parameters Selection: An Average-Case Approach*, URL, April 2023.

[33] Abdelrahaman Aly, Tomer Ashur, Eli Ben-Sasson, Siemen Dhooghe, Alan Szepieniec. *Design of Symmetric-Key Primitives for Advanced Cryptographic Protocols*, URL, April 2019.

[34] Gregory V. Bard. *Algebraic Cryptanalysis*, URL, 2009.

[35] David Forney. *Introduction to finite fields*, URL, September 2016.

[36] Christopher J. Shallue. *Permutation Polynomials of Finite Fields*, URL, May 2012.

[37] Jérémy Jean. *TikZ for Cryptographers*, URL, 2016.

[38] Eli Biham, Adi Shamir. *Differential Cryptanalysis of DES-like Cryptosystems*, URL, January 1991.

[39] Claude E. Shannon. *Communication Theory of Secrecy Systems*, URL, October 1949.

[40] Mitsuru Matsui. *Linear Cryptanalysis Method for DES Cipher*, URL, 1993.

[41] Thomas Jakobsen, Lars R. Knudsen. *The interpolation attack on block ciphers*, URL, February 1996.

[42] Lars R. Knudsen. *Truncated and higher order differentials*, URL, 1994.

[43] Xuejia Lai. *Higher Order Derivatives and Differential Cryptanalysis*, URL, 1994.

[44] Maria Eichlseder, Lorenzo Grassi, Reinhard Lüftenegger, Morten Øygarden, Christian Rechberger, Markus Schofnegger, Qingju Wang. *An Algebraic Attack on Ciphers with Low-Degree Round Functions: Application to Full MiMC*, URL, February 2020.

[45] Maria Eichlseder, Marcel Nageler, Markus Schofnegger. *CRYPTANALYSIS*, URL, 2024.

[46] Cheng-Long Li, Kai-Yi Zhang, Xingjian Zhang, Kui-Xing Yang, Yu Han, Su-Yi Cheng, Hongrui Cui, Wen-Zhao Liu, Ming-Han Li, Yang Liu, Bing Bai, Hai-Hao Dong, Jun Zhang, Xiongfeng Ma, Yu Yu, Jingyun Fan, Qiang Zhang, Jian-Wei Pan. *Device-independent quantum randomness–enhanced zero-knowledge proof*, URL, November 2023.

[47] Kalikinkar Mandal. *Cryptographic Pseudorandom Noise Generators for Lattice-based Cryptography and Differential Privacy*, URL, August 2022.

[48] Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, Elaine Barker, Stefan Leigh, Mark Levenson, Mark Vangel, David Banks,Alan Heckert, James Dray, San Vo. *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*, URL, April 2010.

[49] Robert G. Brown. *DieHarder: A Gnu Public License Random Number Tester*, URL, 2006.

[50] George Marsaglia. *Diehard Battery of Tests of Randomness*, URL, 1995.

[51] Arnab Roy, Matthias Steiner. *An Algebraic System for Constructing Cryptographic Permutations over Finite Fields*, URL, April 2022.

[52] Tianren Liu, Angelos Pelecanos, Stefano Tessaro, Vinod Vaikuntanathan. *Layout Graphs, Random Walks and the t-wise Independence of SPN Block Ciphers*, URL, January 2024.

[53] Anastasiya Gorodilova. *On a remarkable property of APN Gold functions*, URL, March 2016.

[54] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, Yuval Yarom. *Spectre Attacks: Exploiting Speculative Execution*, URL, 2019.

[55] Boru Chen, Yingchen Wang, Pradyumna Shome, Christopher W. Fletcher, David Kohlbrenner, Riccardo Paccagnella, Daniel Genkin. *GoFetch: Breaking Constant-Time Cryptographic Implementations Using Data Memory-Dependent Prefetchers*, URL, 2024.

[56] Jonathan L. Gross. *Combinatorial Methods with Computer Applications*, URL, 2008.