University of Peloponnese
Department of Informatics

Post Quantum Cryptography

Undergraduate Thesis

Author:
Konstantinos Spalas

Supervisor:
Nicholas E. Kolokotronis
Associate Professor

Date

Synopsis

The goods of Confidentiality, Integrity and data Availability are fundamental rights for every human. Securing these fundamentals is one of my major aspects in my life. As we live in a modern, digital, high quality and state-of-the-art environment I got inspired to dive deeply into the principals that digital security is relied. That internal need drove me to develop a relative undegraduate thesis. The help and inspiration from associate professor, mr. Nickolas Kolokotronis was pretty determining.

Cryptography, as a method, has roots from the very ancient years when sensitise information had to be hidden during their transmission. With such practises, people managed to conserve the aforementioned fundamentals. The first attempt of a machine that ciphers a message is aged in ancient Sparta . That machined was called Spartan baton. Later, Caesar came up to a mathematic method that relied in letter sifting. He was able to send messages to his generals without been broken.

It is true that when there an action produces a reaction. So, cryptography has its own rivalry which is cryptanalysis and vice versa. As long the latter become more efficient and sophisticated the need for more effective methods of cryptography is a must. While humanity constantly evolves the field of cryptology, which is both cryptography and cryptanalysis, it converted as a branch of the science of maths. Very advanced math techniques are the basis for both legs of this science. Apart the power of mathematics, computer hardware plays key role in modern cryptology. A state-of-the-art hardware are the quantum computers which states the future of cryptography uncertain, due to their high computational power. Thus, efforts to break ciphers will not be that inefficient anymore, in respect of time and space.

This thesis deals with a system that is able to resist to quantum attacks in an effective matter and is known as the McEliece cryptosystem. Furthermore, the other significant part of this thesis mentions algorithms that attack efficiently the McEliece cryptosystem, using non-quantum computers. It is proven that these sophisticated decoding algorithms are still more attractive comparing to these were quantum computers may use in the future.

In order someone starts studying the McEleiece cryptosystem, he must have already understood its backbone. The Binary Goppa codes is the basis where Robert J. McEliece built this well known cryptosystem. At the upcoming chapters is deployed an effort to summarize all the coding theory components. So, someone be well briefly informed about the structure of the Binary Goppa codes.

Apart from the pros of the McEliece cryptosystem there are also some cons. We will see that the major drawback is its key length. In the final section of the thesis are projected some efforts of well known scientists that try to overcome the key

size problem. One of these efforts listed proposes a hardware implementation of the McEliece system.

Konstantinos Spalas
Tripolis
September 2022

Πρόλογος

Τα αγαθά της Εμπιστευτικότητας, Ακεραιότητας και της Διάθεσης της πληροφορίας είναι μερικά από τα θεμελιώδη δικαιώματα του ανθρώπου από τη στιγμή που άρχισε να δημιουργεί κοινωνίες. Η διασφάλιση αυτών των δικαιωμάτων αντικατοπτρίζουν ένα σημαντικό κομμάτι της ζωής μου. Καθώς στη σημερινή εποχή ζούμε σε ένα μοντέρνο, ψηφιακό και υψηλής ποιότητας περιβάλλον, εμπνέομαι βαθύτατα να καταπιαστώ με όλες τις επιστημονικές αρχές που πλαισιώνουν τη διασφάλιση των ανωτέρω αρχών στο ψηφιακό περιβάλλον. Αυτή η ανάγκη όπου χρόνια κυοφορούνταν έγινε πραγματικότητα, με την αμέριστη βοήθεια του Αναπληρωτή καθηγητή , κ. Νικόλα Κολοκοτρώνη.

Η κρυπτογραφία, σαν μέθοδος, γεννήθηκε από την ανάγκη να μεταδοθούν ευαίσθητες πληροφορίες. Οι περισσότερες από αυτές αφορούσαν στρατιωτικό περιεχόμενο. Η πρώτη προσπάθεια χρονολογείται στην αρχαία Σπάρτη. Η συσκευή που χρησιμοποιήθηκε ονομάστηκε Σπαρτιάτική σκυτάλη. Αργότερα, ο Ιούλιος Καίσαρας εισήγαγε μια πιο εκλεπτυσμένη μέθοδο, βασιζόμενη περισσότερο στα μαθηματικά. Κατάφερε με την μεταγωγή του αλφαβήτου προς μια κατεύθυνση, κάτι που αποτελούσε και το κλειδί της κρυπτογράφησης, να αποστέλλει μη αναγνώσιμα μηνύματα. Η εξέλιξη, όπως είναι αναπόφευκτο, συνεχίζεται μέχρι σήμερα.

Είναι αλήθεια πως η δράση φέρνει αντίδραση. Η κρυπτογραφία έχει το αντίπαλό δέος της που είναι η κρυπτανάλυση. Οι δυο αυτοί κλάδοι αποτελούν την επιστήμη της κρυπτολογίας. Όσο η κρυπτανάλυση γίνεται πιο αποτελεσματική τόσο πρέπει η κρυπτογραφία να γίνεται αποτελεσματικότερη. Σε αυτή την αέναη προσπάθεια έχουν συμβάλει τα τελευταία χρόνια οι ηλεκτρονικοί υπολογιστές. Όσο θα μεγαλώνει η επεξεργαστική τους ισχύ, με αποκορύφωμα της έλευση των κβαντικών υπολογιστών, τόσο θα καθίσταται επιτακτική η ανάγκη να εισάγουμε επαρκείς μεθόδους κρυπτογραφίας.

Η παρούσα διπλωματική εργασία, που υλοποιείται στα πλαίσια των δεύτερων προπτυχιακών μου σπουδών, προσπαθεί να περιγράψει ένα κρυπτογραφικό σύστημα το οποίο έχει αποδειχτεί θεωρητικά πως μπορεί να αντισταθεί σε επιθέσεις από αλγόριθμους κβαντικών υπολογιστών. Αυτό το σύστημα ονομάζεται κρυπτοσύστημα McEliece προς τιμής του εφευρέτη του Robert J. McEliece.

Επιπρόσθετα, η διπλωματική αναφέρεται και σε ορισμένους συμβατικούς (μη-κβαντικούς) αλγόριθμους κρυπτανάλυσης ενάντια στο προαναφερόμενο κρυπτογραφικό σύστημα. Μελέτες αποδεικνύουν πως αυτοί είναι αποδοτικότεροι συγκρινόμενοι με αυτούς που θα υλοποιηθούν για τα κβαντικά συστήματα. Άλλωστε, δεν είναι τυχαίο πως στο διαγωνισμό του NIST για την μετακβαντική εποχή, το εν λόγω κρυπτοσύστημα ήταν φιναλίστ.

Εντούτοις, εκτός από τα προτερήματα του προαναφερόμενου κρυπτοσυστήματος, τα οποία θα αναλυθούν αργότερα, σε σχέση με τους ήδη υπάρχοντες αλγόριθμους, υπάρχουν και αρνητικά χαρακτηριστικά που τον καθιστούν, στην παρούσα φάση, σχετικά δύσχρηστο. Ένα από αυτά είναι το μέγεθος του δημόσιου κλειδιού. Όμως, γίνονται αρκετές προσπάθειες από τους επιστήμονες του χώρου ώστε να μειωθεί το μέγεθος αυτό.

Προκειμένου κάποιος να κατανοήσει το κρυπτοσύστημα McEliece πρέπει πρωταρχικά να κατανοήσει βασικές αρχές της θεωρίας κωδίκων (coding theory). Τούτο το σύστημα βασίζεται στους κώδικες Goppa οι οποίοι με τη σειρά τους βασίζονται στην θεωρία κωδίκων. Τα σημαντικότερα σημεία που είναι κομβικά για την κατανόηση των Goppa κωδίκων περιγράφονται με πληρότητα σε αυτή την εργασία.

Contents

*

List of Figures

List of Tables

List of Algorithms

Code Listings

CHAPTER 1

Introduction

The historical origins of coding theory are in the problem of reliable communication over noisy channels. Claude Shannon, which is said to be the father of communication, in the introduction to his classic paper, 'A Mathematical Theory of Communication' wrote in 1948 that 'The fundamental problem of communication is that of reproducing at one point either exactly or approximately a message selected at another point'. Error-correcting codes are widely used in applications such as returning pictures from deep space, design of numbers on Debit/Credit cards, ISBN on books, telephone numbers generation, cryptography and many more. It is fascinating to note the crucial role played by mathematics in successful deployment of those. The progress of cryptography is closely related with the development of coding theory.

In late 1970s, coding theory started shaping public key cryptography. Some of these cryptosystems, like RSA, are proved to have substantial cryptographic strength which can achieve high security standards. Though algorithms developed by Peter Shor and Lov Grover may promise to reveal RSA componets using quantum computers, there are also another cryptographic algorithms that sustain.

Based on error-correcting codes and the difficult problem of decoding a message with random errors, the security of McEliece cryptosystem does not depend on the difficulty of factoring integers or finding the discrete logarithm of a number like in RSA, ElGamal and other well known cryptosystems. The security in McEliece cryptosystem lies on the ability of recovering plaintexts from ciphertexts, using a hidden error-correcting code, which the sender initially garbles with random errors. Quantum computers do not seem to give any significant improvements in attacking code-based systems, beyond the improvement in brute force search possible with Grovers algorithm. Therefore McEliece encryption scheme is one of the interesting code-based candidates for post-quantum cryptography. In the originally proposed system, a codeword is generated from plaintext message, k bits, by using a permuted and scrambled generator matrix of a Goppa code of length n, capable of correcting t errors.

On the other hand, there is being developed a batch of algorithms that is able to break ciphers produced by the McEliece cryptosystem. Such algorithms belong

at the Information Set Decoding (ISD) method. Information-set decoding is a probabilistic decoding strategy that essentially tries to guess k correct positions in the received word, where k is the dimension of the code. A codeword agreeing with the received word on the guessed position can easily be computed, and their difference is one possible error vector. Thus, the major effort of ISD is to reveal these errors. When transmmiting a message it is probable to rise such errors due to noise. In case of the McEliece cryptosystem, this noise in intentionally added.

The initial concept of this thesis is to give an overall knowledge of the aforementioned mathematic models. Hence, in chapter 2 is projected some representatives of the modern cryptographic methods, ending in the post quantum era. In chapter 3 are broken down all the components that compose the coding theory, which is the basis of the present star cryptosystem and descripted in 4. Furthermore, chapter ?? lists some methods to attack the McEliece cryptosystem. Finally, from chapter 6 is reported the nowadays research status of Post Quantum Security (PQS).

Additionally, in order to come to conclusions about the effectivity of the ISD algorithms and their correlation in respect of time execution, it has been en effort to develop in Python some representatives. Some of them are Prange's alg (or Plain-ISD), Stern's alg and finally ball-collision alg. The source code is listed in https://github.com/konnnGit/mceliece-isd/tree/final, which contains also a McEliece implementation.

CHAPTER 2

Introduction to Cryptography

Whenever we transmit sensitive information over a public channel, there is potential that an eavesdropper could intercept this message and steal this sensitive information. Fortunately, modern-day transmissions are protected using the principles of cryptography, the study of transmitting secure messages that may be intercepted by an adversary. The main idea behind cryptography is that the sender selects a message that he or she would like to transmit, applies some sort of encryption process, and transmits this encrypted message across the channel. The receiver obtains the encrypted message, also referred to as a ciphertext, then uses a known decryption process to recover the senders original message. If an adversary intercepts an encrypted message, he or she will be unable to recover the original message without knowledge of the secret decryption process. The encrypting and decrypting processes usually involve the use of a piece of secret information, known as a private key, which can be used to perform these tasks. One should think of encryption and decryption as inverse operations of one another.

In order to further discuss cryptography, a few important distinctions and clarifications need to be made. Cryptography is the study of transmitting secure messages, and a cryptosystem is an implementation of a particular cryptographic algorithm. Cryptanalysis refers to the study of finding and exploiting weaknesses in a particular cryptosystem. These are commonly referred as attacks against a cryptosystem.

Mainly, cryptosystems are typically classified as being one of two types: private or public key. There are benefits and drawbacks to each type of cryptosystem, and we will briefly discuss each below. In the following discussion, we will suppose that two friends, Alice and Bob, who would like to communicate securely.

2.1 Hash Function Cryptography

A cryptographic hash function (CHF) is a mathematical algorithm that maps data of an arbitrary size (often called the "message") to a bit array of a fixed size (the "hash value", "hash", or "message digest"). It is a one-way function, that is, a function for which it is practically infeasible to invert or reverse the computation. Ideally, the only way to find a message that produces a given hash is to attempt a

brute-force search of possible inputs to see if they produce a match, or use a rainbow table of matched hashes. Cryptographic hash functions are a basic tool of modern cryptography.

Hash functions are also used for implementing digital signature schemes. A digital signature is working like an ordinary signature. They both have the characteristic of being easy for a user to produce, but difficult for anyone else to forge. Digital signatures can also be permanently tied to the content of the message being signed. They cannot then be 'moved' from one document to another, for any attempt will be detectable.

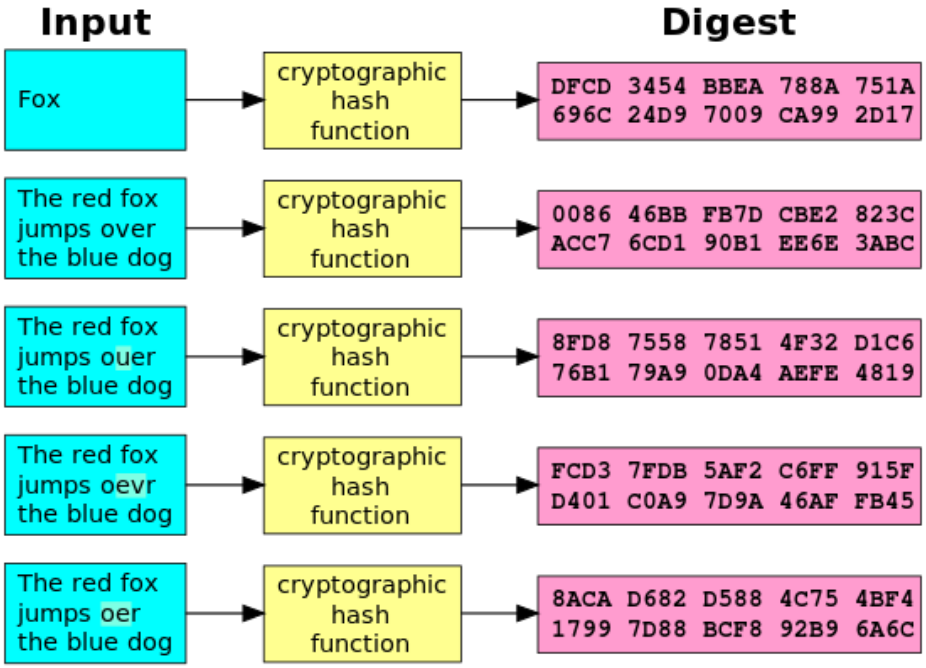

Figure 2.1: Hash Function Cryptography

2.2 Private (or Symmetric) Key Cryptography

Symmetric key cryptography refers to encryption methods in which both the sender and receiver share the same key (or, less commonly, in which their keys are different, but related in an easily computable way). This was the only kind of encryption publicly known until June 1976 [3].

Symmetric key ciphers are implemented as either block ciphers or stream ciphers. A block cipher enciphers input in blocks of plaintext as opposed to individual characters, the input form used by a stream cipher.

The Data Encryption Standard (DES) and the Advanced Encryption Standard (AES) are block cipher designs that have been designated cryptography standards by the US government. Though, DES's designation was finally withdrawn after

the AES was adopted [4]. Despite its deprecation as an official standard, DES (especially its still-approved and much more secure the triple-DES variant) remains quite popular. It is used across a wide range of applications, like ATM encryption [5]. Many other block ciphers have been designed and released, with considerable variation in quality. Many, even some designed by capable practitioners, have been thoroughly broken.

Stream ciphers, in contrast to the *block* type, create an arbitrarily long stream of key material, which is combined with the plaintext bit-by-bit or character-by-character, somewhat like the $one-time-pad$. In a stream cipher, the output stream is created based on a hidden internal state that changes as the cipher operates. That internal state is initially set up using the secret key material. RC4 is a widely used stream cipher [6]. Block ciphers can be used as stream ciphers by generating blocks of a keystream (in place of a Pseudorandom number generator) and applying an XOR operation to each bit of the plaintext with each bit of the keystream.

In private key cryptography, Alice and Bob share a single piece of information (known as the $private-key$) which is the secret, necessary to perform both the encryption and decryption processes. In the case of private key cryptography, a naive observer has virtually no knowledge of the cryptosystem. The only information available to an attacker is intercepted ciphertexts.

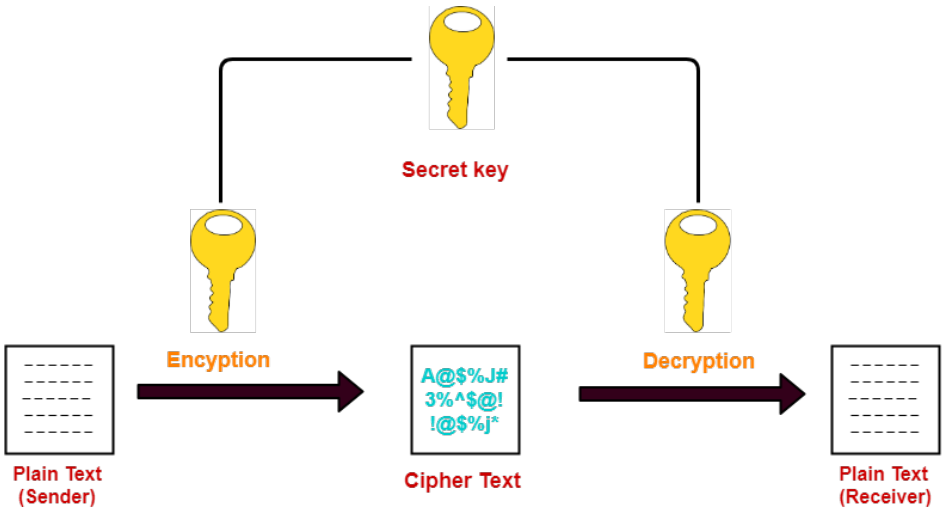

Figure 2.2: Symmetric Key Cryptography

2.2.1 Example (One-Time-Padding)

In one example of a $One-Time-Padding$ scheme, Alice and Bob both agree on a single, random n-bit binary vector p (known as the pad). In this case, p is the private key shared by Bob and Alice. When Alice would like to transmit a message to Bob, she performs a $modulo-2$ addition between p and her message m and transmits the

result r to Bob.

$$r = m \oplus p$$

Note that binary $modulo - 2$ addition is the same as XOR (exclusive OR) .

This addition constitutes the entirety of the encryption process. When Bob receives Alices encrypted message, he uses the same pad p and performs the same addition of p to the received message. What it comes is

$$r \oplus p = (m \oplus p) \oplus p = m \oplus (p \oplus p) = m \oplus 0 = m$$

, which is the original message. Since only Alice and Bob know the secret pad, any third party that intercepts the encrypted message will have a difficult time deducing the original message.

One of the primary disadvantages to private key cryptography relates to the difficulty of keeping the private key secret or use more than one an synchronize them. In order to protect the cryptosystem from attacks, the private key is often frequently changed, and the process of agreeing on a private key may need to take place in person. Furthermore, increasing the number of users in this cryptosystem also increases the chances that the system will be broken. Thus, private key cryptosystems do not scale well. These difficulties are not present in public key cryptography.

2.3 Public (or Asymmetric) Key Cryptography

Public key cryptography, also known as asymmetric key cryptography, takes another approach to the process of encrypting and decrypting. In public key cryptography each one , Alice and Bob for our paradigm, maintain a distinct private key and also a distinct public key. A public key is a piece of information that is published for all parties to see. Thus, Bob and Alice each publish a public key, but they also each keep a single piece of information secret. Only Alice knows her private key and only Bob knows his private key. Note that an individuals public and private key are typically related in some way that facilitates and enables the encryption and decryption process.

Suppose Alice wishes to send Bob a message. Alice begins by looking up Bob's public key. Alice then uses this public key to encrypt her message and transmits the result to Bob. Bob receives Alice's transmitted message and uses his secret private key to decrypt the encrypted message and recover Alices original message. Notice that the information available to an attacker has increased substantially. The attacker now has knowledge of a public key (which is related to the private key used for decryption) in addition to the ciphertexts. In order to ensure security, it must therefore be difficult to produce the private key from the public key. A well-known implementation of public key cryptography, RSA, is provided as an example.

Public-key is also used in digital signature schemes. There are two algorithms: one for signing, in which a secret key is used to process the message (or a hash of the message, or both), and one for verification, in which the matching public key is

used with the message to check the validity of the signature. RSA and DSA are two of the most popular digital signature schemes. Digital signatures are central to the operation of public key infrastructures and many network security schemes.

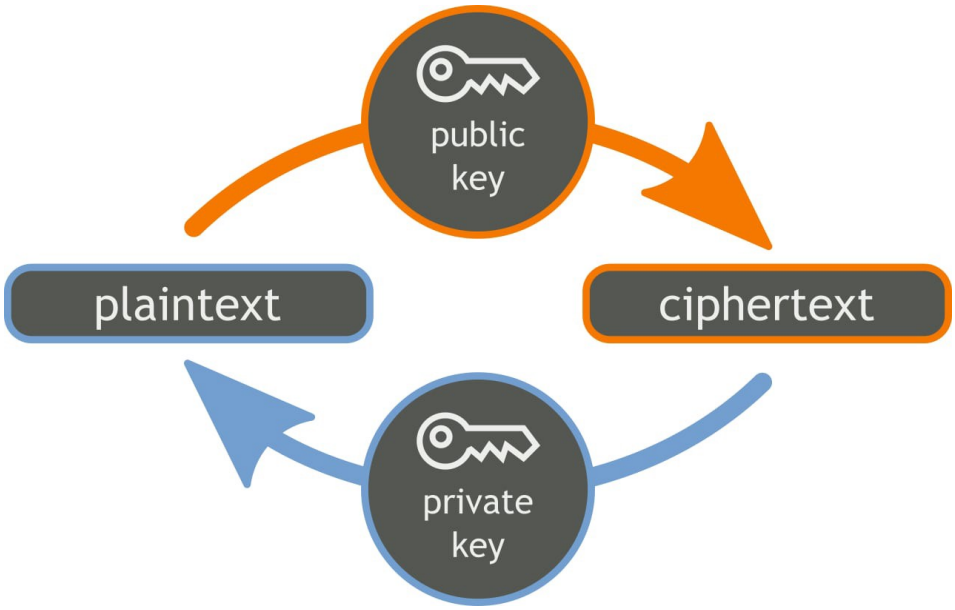

Figure 2.3: Public Key Cryptography

2.3.1 Example (RSA)

RSA is named after its inventors, Rivest, Shamir, and Adelman [1]. In order for Bob to set up his portion of an RSA cryptosystem, he begins by selecting two large, prime integers, p and q, and computes the product $n = pq$. Product n is used as the modulus for both the public and private keys and is part of the public key. Its length, usually expressed in bits, is the key length. Bob then picks an e coprime for the function $\phi(n) = (p-1)(q-1)$, such that $1 < e < \phi(n)$, where ϕ(n) is Eulers function. Bob then publishes as the public key, the pair (n, e) but keeps the pair (p, q) as his private key.

When Alice wants to transmit a secure message to Bob, she begins by looking up Bob's public key from a database containing everyone's public keys. She uses the information (n, e) and encrypts her message m (now represented as an integer instead of binary vector) as

$$me \bmod (n)$$

Since it is a hard problem to undo modular exponentiation [2], a third party that intercepts this encrypted message will have a hard time recovering the original message without knowledge of p and q. When Bob receives the message, he uses his secret knowledge of p and q to 'undo' the exponentiation. RSA is considered secure due to the hardness of factoring n into p and q to compute ϕ(n).

The advantage of public key cryptosystems key management is relatively simple. There is no need to meet to exchange private keys, and the system scales well because new members can be added without impacting existing users. Another noteworthy advantage is the ability to 'sign' messages to ensure the sender is who he/she claims. One downside to public key cryptography is that an attacker has more information that can be used for cryptanalysis.

It is not the willing of this thesis to expand more information details about RSA and Public Key Cryptography (PKC). An interested reader may refer to [1], for more details.

2.4 Post Quantum Cryptography

2.4.1 The Quantum Vulnerability of the RSA System

In the popular RSA system, the public key is the product $n = pq$ of two secret prime numbers p and q. The security of RSA relies critically on the difficulty of finding the factors p, q of n. However, in 1994, Shor introduced a fast quantum algorithm [2] to find the prime factorization of any positive integer n.

In general, suppose the order r of some integer x: $x < n$. Suppose further that x is even. This is necessary in order that $x^{r/2}$ be an integer. By definition then, r is the smallest factor of x.

Shor's algorithm, which is designed to take advantage of the inherent potential of quantum, in contrast to classical computers, exploits a factorization method that differs from the trivial, for large key number q.

Then, compute the great common divisor between $x^{r/2} - 1$ and n, denoted as $\gcd(x^{r/2} - 1, n)$. The Euclidean algorithm takes polynomial time to compute the gcd between two numbers. Since the multiplication of the two numbers $\gcd(x^{r/2} - 1)$ $\gcd(x^{r/2} + 1) = x^r - 1 \equiv 0 \ (mod\ n)$, then $\gcd(x^{r/2} - 1)$ and $\gcd(x^{r/2} + 1)$ will be two factors of n. This procedure fails only if r is odd, in which case $r/2$ is not in integer or if $x^{r/2} \equiv -1 \ (mod\ n)$

The probability that a randomly selected $x < n = pq$ and coprime to n will have an even order r, satisfying the aforementioned equations is at least $1 - \frac{1}{2^{k-1}}$, where k is the is the number of distinct odd prime factors of n, and at most is $\frac{1}{2}$ [15]. Calculating the gcd of a pair of large numbers using classical computers is a straightforward procedure requiring negligible computing time. Therefore, the feasibility of factoring a large $x < n = pq$ via the procedure described depends primarily on the feasibility of determining the order r of x (mod n), for arbitrarily selected x. With classical computers this determination requires solving the discrete log problem.

2.4.2 Post Quantum Alternatives

With quantum computers, determining r, and thereby factoring n, becomes feasible using the periodicity property of the sequence f_j, $j = 1, 2, 3, ..$, with the order r be the smallest j.

There has been some research into analysing and optimizing the exact costs of Shor's algorithm . For example, a variant of Shor's algorithm, by Beauregard [14], uses $2n + 3$ qubits in a depth of $O(n^3)$, if $n = pq$ fits into n bits. An order-finding circuit using elementary gates and less than $2n + O(1)$ qubits is not ruled out yet, but it seems that a different method would have to be used for modular multiplication to get such a circuit.

Although quantum computing is still in its nascent stage, its radical progress jeopardizes the most well-known public-key encryption schemes. Such schemes are fundamental for todays digital landscape and predominantly the security of the internet, due to their potential for providing security in non-secure communications channels that allow for exchanging secret government documents, industrial trade secrets, military communications or financial and medical records, inter alia. Currently, secure and reliable data communication in IoT and other Distributed Ledger Technologies (DLTs) is also based on public-key cryptosystems such as Elliptic Curve Cryptosystem (ECC) [9].

A McEliece cryptosystem is a public key cryptosystem that leverages error-correcting codes as a method of encryption. Initially proposed by Robert J. McEliece in [7], this cryptosystem has a number of advantages which will be discussed later in detail. Among these advantages is the ability of McEliece cryptosystem to resist

cryptanalysis, especially in a quantum computer setting. This property is especially of interest because some modern day public key cryptosystems, including the aforementioned RSA, can be broken using quantum computers. It is for this reason that McEliece cryptosystem are important in both theory and real world application. But before we proceed to a thorough discussion for the McEliece cryptosystem, we must first discuss and understand the notion of error-correcting codes.

The security of code-based cryptographic constructions relies on the conjectured intractability of the Computational Syndrome Decoding (CSD) problem for random linear codes. The CSD problem is one of the most fundamental combinatorial problems in the coding theory . Besides its relevance to the security of code-based cryptographic constructions, the average-case hardness of the CSD problem or equivalently the hardness of decoding random linear codes has been of profound importance in coding theory, in general. This sufficiently motivates the main objective of this work.

CHAPTER 3

Coding Theory

In 1948 Claude Shannon managed to excel the communication over noisy transmission channels [16]. His central theme was that if the signaling rate of the system is less than the channel capacity, reliable communication can be achieved if one chooses proper encoding and decoding techniques. The design of good codes and of efficient decoding initiated by Hamming, Golay, and others in the late 1940s, has since occupied the energies of many researchers. Much of this work is highly mathematical and a thorough understanding requires an extensive background in modern algebra and probability theory. This requirement has impeded many engineers and computer scientists who are interested in applying these techniques to practical systems. One of the purposes of this thesis is to present the essentials of this highly complex material in such a manner that it can be understood and applied with only a minimum of mathematical background.

Coding research in the 1950s and 1960s was devoted primarily to developing the theory of efficient encoders and decoders. In 1970 the first author published a book entitled 'An Introduction to Error-Correcting Codes', which presented the fundamentals of the previous two decades of work covering block codes. The approach was to explain the material in an easily understood manner, with the minimum mathematics. Then, in 1983, the authors published the first edition of this book. The approach again stressed the fundamentals of coding. In addition, new material on many of the practical applications of coding developed during the 1970s was introduced. Other major additions included a comprehensive treatment of the error-detecting capabilities of block codes and an emphasis on soft decoding methods for convolutional codes.

In the 1980s and 1990s, the coding field exploded with new theoretical developments, several of which have had significant practical consequences. Three of these new developments stand out in particular:

(i) The application of binary convolutional and block codes to expanded (nonbinary) modulation alphabets

(ii) The development of practical soft decoding methods for block codes

(iii) The discovery of soft-input, soft- output iterative decoding techniques for block and convolutional codes

These new developments have revolutionized the way coding is applied to practical systems, affecting the design of high-speed data modems, digital mobile cellular telephony, satellite and space communications, and high-density data storage, etc.

During the next paragraphs will be an introduction to the basic elements that the coding theory is consisted of. Most of these elements are based on algebra.

3.1 Groups

Let G be a set of elements. Let a binary operation $*$ on G, be a rule that assigns to each pair of elements a and b, which leads to a uniquely defined third element $c = a * b$, also belong in G. When such a binary operation $*$ is defined on G, we say that G is closed under that operation.

For example, let G be the set of all integers and let the 'binary' operation on G be the real addition $+$. We all know that, for any two integers i and j in G, $i + j$ is a uniquely defined integer in G. Hence, the set of integers is closed under real addition. In other words, a close operation of two distinct elements in a group leads to another element in this group.

A set G, on which a binary operation $*$ is defined, is called a group if the following conditions are satisfied:

(i) The binary operation $*$ is *associative* in G, which means that for any a, b and c in G

$$a * (b * c) = (a * b) * c. \tag{3.1}$$

(ii) G contains an element e such that, for any a in G

$$a * e = e * a = a. \tag{3.2}$$

This element e is called the identity element of G. It is unique for the group.

(iii) For any element a in G, there exists another element a' in G such that

$$a * a' = a' * a = e \tag{3.3}$$

The element a' is called the inverse of a (a is also called the inverse of a'). Inverses are unique each other for a group.

A group G is said to be *commutative* if, for any a, b and c in G, its binary operation $*$ also satisfies

$$a * b = b * a \tag{3.4}$$

For example, let the set of two integers G={ 0,1 }. Let also define the binary operation ⊗ on G as follows:

$$0 \otimes 0 = 0$$
$$0 \otimes 1 = 1$$
$$1 \otimes 0 = 1$$
$$1 \otimes 1 = 0$$

This binary operation is called modulo-2 addition. The set G={ 0,1 } is a group under modulo-2 addition. It follows from the definition of modulo-2 addition ⊗ that G is closed under ⊗ and ⊗ is commutative.

We can easily check that ⊗ is associative: The element 0 is the identity element. The inverse of 0 is itself, and the inverse of 1 is also itself. Thus, G together with ⊗ is a commutative group.

The number of elements in a group is called the *order* of the group. A group of finite order is called a finite group. For any positive integer m, it is possible to construct a group of order m under a binary operation that is very similar to real addition.

3.2 Fields

Group's concepts used to introduce another algebraic system, called *field*. Roughly speaking, a field is a set of elements in which we can perform addition, subtraction, multiplication and division without leaving the set. Addition and multiplication must satisfy the commutative, associative, and distributive laws. More formally, a field F is:

(i) A commutative group under addition +. The identity element with respect to addition is called the zero element or else, the *additive identity* of F and is denoted by 0.

(ii) The set of nonzero elements in F is a commutative group under multiplication · . The identity element with respect to multiplication is called the unit element or else the *multiplicative identity* of F and is denoted by 1.

(iii) Multiplication is *distributive* over addition. So, for any three elements a,b and c in F :

$$a \cdot (b + c) = a \cdot b + a \cdot c \tag{3.5}$$

It follows from the definition that a field consists of at least two elements, the additive identity and the multiplicative identity. A field of two elements does exist. The number of elements in a field is called the order of the field. A field with a finite number of elements is called a finite field. In a field, the additive inverse of

an element a is denoted by $-a$ and the multiplicative inverse of it is denoted by a^{-1}, provided that $a \neq 0$. Subtracting a field element b from another field element a is defined as adding the additive inverse of b to a. So,

$$a - b = a + (-b)$$

Consider the set 0, 1 together with modulo-2 addition and modulo-2 multiplication. This field is usually called a binary field and is denoted by F(2). The binary field F(2) plays an important role in coding theory and is widely used in digital computers and digital data transmission (or storage) systems.

Table 3.1: Modulo 2 addition

+	0	1
0	0	1
1	1	0

Table 3.2: Modulo 2 multiplication

.	0	1
0	0	0
1	1	0

Let p be a prime. The set of integers $\{0, 1, 2, ..., p-1\}$ is a commutative group under modulo-p addition. Because this field is constructed from a prime, p, it is called a prime field and is denoted by F(p). For $p = 2$, we obtain the binary field F(2). For any prime p, there exists a finite field of p elements. In fact, for any positive integer m, it is possible to *extend* the prime field F(p) to a field of p^m elements, which is called an extension field of F(p) and is denoted by

$$F(p^m)$$

Furthermore, it has been proved that the order of any finite field is a power of a prime. Finite fields are also called Galois fields, in honor of Evariste Galois, their discoverer and are denoted as

$$GF(p^m) \tag{3.6}$$

A large portion of algebraic coding theory, code construction, and decoding is built around finite fields.

In general, we can construct codes with symbols from any Galois field GF(q), where q is either a prime p or a power of a prime p. Codes with symbols from the binary field GF(2) or its extension, like 3.6, are most widely used in digital data transmission and storage systems because information in these systems is universally coded in binary form for practical reasons.

Now, let α be a nonzero element in GF(q). Since the set of nonzero elements of that field is closed under multiplication, the following powers of a are represented as

$$\alpha^1 = \alpha$$
$$\alpha^2 = \alpha \cdot \alpha$$
$$\alpha^3 = \alpha^2 \cdot \alpha = \alpha \cdot \alpha \cdot \alpha$$
$$...$$

Because GF(q) has finite number of elements, the powers of a given element in the finite field must lead us to another element in the field. Therefore, at some point in the sequence of powers of α there must be a repetition. So, there must exist distinct positive integers m, k and n such that

$$m > k : \alpha^m = \alpha^k \Rightarrow \alpha^{m-k} = 1 \tag{3.7}$$

The 3.7 implies that there is a positive integer n:

$$\alpha^n = 1 \tag{3.8}$$

The smallest integer n that satisfies 3.8 is called the *order* of the field. Threfore, the sequence

$$\alpha, \alpha^2, \alpha^3, ..., \alpha^n \tag{3.9}$$

repeats after $\alpha^n = 1$. Though, the powers of the sequence in 3.9 are distinct between them.

In a finite field GF(q), a nonzero element α is said to be primitive if the order of α is $q - 1$. The powers of a primitive element generate all the nonzero elements of GF(q). Every finite field has a primitive element.

3.3 Polynomials over the binary GF(2)

Considering with the manner of the chapter 3.2 and constructing fields with a finite number of elements, we can define polynomials over a filed using these elements. Let introduce a polynomial $f(x)$ over the binary Galois field GF(2), with the variable x. By saying 'over the binary Galois field GF(2)' means that the coefficients f_i are from the GF(2). The form of that will be

$$f(x) = f_0 + f_1 x + f_2 x^2 + ... f_n x^n \tag{3.10}$$

, where $f_i = 0$ or 1 for $0 \leq i \leq n$.

The degree of a polynomial is the largest power of x with a nonzero coefficient. For instance, there are two polynomials, over GF(2), with degree 1, the

$$x \tag{3.11}$$
$$x + 1 \tag{3.12}$$

and there are four polynomials with degree 2, the

$$x^2 \tag{3.13}$$
$$x^2 + 1 \tag{3.14}$$
$$x^2 + x \tag{3.15}$$
$$x^2 + x + 1 \tag{3.16}$$

In general, there are 2^n polynomials with degree n.

Polynomials over GF(2) can be added, subtracted, multiplied and divided with other polynomials over the field. For example, let us use the modulo-2 addition for the polynomials 3.12 and 3.16. It will be

$$\begin{aligned} f(x) &= x + 1 \\ g(x) &= x^2 + x + 1 \\ f(x) + g(x) &= x + 1 + x^2 + x + 1 = 1 \cdot x + 1 + 1 \cdot x^2 + 1 \cdot x + 1 = x^2 \end{aligned}$$

because adding the coefficients in mod 2 returns zero (0). The same is applied when multiply. Generally we can use the tables 3.1 and 3.2 for the coefficients.

Also, considering same polynomials $g(x)$ and $f(x)$ we also introduce the polynomial

$$s(x) = 1 + x^2 + x^3 + x^4$$

It can be easily seen that $s(1) = 0$. Then we can say that 1 is a *root* of the $s(x)$. Any polynomial, in real numbers, that has a root r, then this is divisible by $x - r$. For the binary field, though, happens the same with the difference that

$$x - r \equiv x + r \tag{3.17}$$

According to 3.17 it is deducted that the $s(x)$ is divisible with $x+1$. In such occasions the remainder $r(x)$ of the division is zero $[r(x) = 0]$. It is also deducted that any polynomial with even number of terms has as root the 1 and then is divisible by $x + 1$.

A polynomial $f(x)$ of degree d over a Galois field is said to be *irreducible* if it is not divisible by any polynomial of degree d' over that field, with $0 < d' < d$. For instance the polynomial $f(x) = x^2$ has the zero as a root because $f(0) = 0$. So, the polynomial $x + 0 = x$ divides $f(x)$ and thus $f(x)$ is not irreducible. On the other hand, $g(x) = x^3 + x + 1$ is irreducible because $g(0) = g(1) \neq 0$, which implies that neither of $0, 1$ are roots of $g(x)$.

An element α of a Galois field GF(p^m) is said to be *primitive* if it can represent all the elements (except of zero) via itself. For example, for the GF(2^3) the number 3 is a primitive element because the powers of it, in modulus-3, produces all the elements of the field which are GF(2^3) = $(1, 2, 3, 4, 5, 6, 7)$.

An irreducible polynomial $p(x)$ of power m, over a Galois field GF(p^m), is said to be *primitive* if the number $n = p^m - 1$ is the smallest integer for which $p(x)$ divides

$x^n + 1$ (in other words, the $p(x)$ is an irreducible factor of $x^n + 1$). We can use a primitive polynomials to represent all the elements of GF(p^m). For instance, we can use the primitive polynomial

$$p(x) = 1 + x + x^4 \tag{3.18}$$

to generate the elements of the $GF(2^4)$. Let assume that the element α, of the field $GF(2^4)$, is root of $p(x)$. So, substituting α to x in 3.18

$$p(\alpha) = 0 \Rightarrow 1 + \alpha + \alpha^4 = 0 \Rightarrow \alpha^4 = 1 + \alpha \tag{3.19}$$

Then, using the equation 3.19 we can construct the $2^4 = 16$ elements of the $GF(2^4)$. The primitive polynomial helps us to represent them as the terms of the known polynomial and furthermore we can represent using a 4-tuple binary mask, as shown in the following table 3.3.

Table 3.3: Representation of $GF(2^4)$ elements using primitive $p(x) = 1 + x + x^4$.

Power	Polynomial	4-tuple
0	0	0000
1	1	1000
α	α	0100
α^2	α^2	0010
α^3	α^3	0001
α^4	$1 + \alpha$	1100
...	...	...
α^{13}	$1 + \alpha^2 + \alpha^3$	1011
α^{14}	$1 + \alpha^3$	1001

3.4 Vectors Space

Let V be a set of elements on which a binary operation called addition, +,is defined. Let F be a field. A multiplication denoted $\cdot$ between the elements in GF(2) and elements in V is also defined. The set V is called a vector space over the field GF(2) if it satisfies the following conditions:

(i) V is a commutative group under addition.

(ii) For any element a in GF(2) and any element w in V, $a \cdot w$ is also an element in V.

(iii) For any elements u and v in V and any elements a and b in F, it is:

$$a \cdot (u + v) = a \cdot u + a \cdot v$$
$$(a + b) \cdot v = a \cdot v + a \cdot b$$

(iv) For any v in V and any a and b in F :

$$(a \cdot b) \cdot v = a \cdot (b \cdot v)$$

(v) Let 1 be the unit element of F. Then, for any v in V, $1 \cdot v = v$.

The elements of V are called *vectors* and the elements of the field F called *scalars*. The addition on V is called a *vector addition* and the multiplication that combines a scalar in F and a vector in V into a vector in V is referred to as *scalar multiplication* or *product*. The additive identity of V is denoted by $\mathbf{0}$.

The vector space over Galois Field GF(2) is very useful because it has a central role in coding theory. Consider an ordered sequence of n components,

$$a_0, a_1, ..., a_{n-1}$$

where each component a_i is an element from an extension of the Binary Galois Field GF(2). This sequence is generally called an n-tuple over GF(2). Because there are two choices for each a_i, we can construct 2^n distinct n-tuples. Let V_n denote this set of 2^n distinct n-tuples over GF(2). We define an addition, +,on V_n as the following: For any $\boldsymbol{u} = (u_0, u_1, ...u_{n-1})$ and $\boldsymbol{v} = (v_0, v_1, ..., v_{n-1})$ in V_n ,

$$\boldsymbol{u} + \boldsymbol{v} = (u_0 + v_0, u_1 + v_1, ..., u_{n-1} + v_{n-1})$$

where each $u_i + v_i$ is actually a modulo-2 addition. As mentioned before, the modulo-2 addition over GF(2) is a closed action. Hence, the $\boldsymbol{u} + \boldsymbol{v}$ is belongs to the V_n and also is commutative group under that addition.

Let us denote the all-zero n-tuple $\mathbf{0} = (0, 0, ..., 0)$. Because modulo-2 addition is commutative and associative, according to table 3.1 we can declare that adding a vector with itself we get the zero vector. In other way, the additive inverse vector of each n-tuple in V_n is itself.

Also, we can define the scalar multiplication of an n-tuple vector in V_n. Let us get an element α from GF(2), that is 0 or 1 and get a vector v from V_n, using the second condition 3.4 above, as follows:

$$a \cdot v = (a \cdot v_1, a \cdot v_2, ..., a \cdot v_n)$$

If a=0, according to the second condition above and the table 3.2, we may declare that the result will be the zero vector. If a=1, according to the same table, would result $1 \cdot v = v$.

Let collect a set of vectors $v_1, v_2, ..., v_k$ in V_n over the GF(2), with $k < 2^n$. These vectors are *linearly dependent* if there exist k scalars $a_1, a_2, ..., a_k$ from GF(2), not all zero ($a_1 = a_2 = ... = a_k \neq 0$), that

$$a_1 \cdot v_1, a_2 \cdot v_2, ..., a_k \cdot v_k = \mathbf{0}$$

The set of vectors above is *linearly independent* if it is not *linearly dependent*, which means that

$$a_1 \cdot v_1, a_2 \cdot v_2, ..., a_k \cdot v_k \neq \mathbf{0}$$

A set of vectors is said to *span* a vector space V if every vector in V is a linear combination of the vectors in the set. In any vector space there exists at least one set of vectors B, that span the space. This set is called a *basis* of the vector space. The number of vectors in a basis of a vector space is called the *dimension* of the vector space.

Let S be a k-dimensional subspace of the vector space V_n of all n-tuples, over GF(2). Let another n-tuples subspace K with dimension $n - k$. Then,

$$dimension(S) + dimension(K) = n$$

For every u in S and v in K, if the dot product $u \cdot v = 0$, then it is said that u and v are *orthogonal* to each other. The subspace K is called the *null* space of S. A subspace is a nonempty space since it, at least, contains the zero n-tuples vector $\mathbf{0}$.

3.5 Matrices over GF(2)

A $k \times n$ matxix G, over GF(2), is a rectangular array with k rows and n columns where each entry g_{ij}, with $0 \leq i < k$ and $0 \leq j < n$ is an element from the GF(2). The i indicates the row of the g_{ij} and j indicates the column. Also, each row of G is an n-tuple over GF(2), and each column is a k-tuple over GF(2).

If the k rows ($k \leq n$) of G are linearly independent, then the 2^k linear combinations of these rows form a k-dimensional subspace of the vector space V_n of all the n-tuples over GF(2). This subspace is called the row space of G.

Let S be the row space of matrix $G_{k \times n}$, over GF(2). Let K be the null space of S. The dimension of K is $n - k$. The h_{ij} with $0 \leq i < k - n$ and $0 \leq j < n$ are the elements of the matrix $H_{(n-k) \times n}$ of the space K. Because the row space S of G is the null space of the row space K of H, we call S the null space of H.

More formally, for any $G_{k \times n}$, over GF(2), with k linearly independent rows, there exists an $H_{(n-k) \times n}$, over GF(2), with $n - k$ linearly independent rows such that for any row g_i in G and any row h_l in H the dot product $g_i \cdot h_l = 0$. The row space of G is the null space of H, and vice versa. In other words, the k row vectors of G are orthogonal with the $n - k$ row vectors of H.

3.6 Linear Block Codes

Because information in most current digital data communication and storage systems is coded in the binary digits 0 and 1, we discuss only the linear block codes with symbols from the binary field GF(2). Linear block codes are defined and described in terms of *generator* (symboled as G) and *parity − check* (symboled as H) matrices.

Assume that the output of an information source is a sequence of the binary digits 0 and 1. In block coding, this binary information sequence is segmented into message blocks of fixed length. Each message block, denoted by $\boldsymbol{u}$, consists of k information digits. There are a total of 2^k distinct messages. The encoder, according to certain rules, transforms each input message $\boldsymbol{u}$ into a binary n-tuple $\boldsymbol{v}$ with n > k. This binary n-tuple $\boldsymbol{v}$ is referred as the codeword (or code vector) of the message $\boldsymbol{u}$. Therefore, there are 2^k codewords and the set of them is called a *block code*. A block code is called *linear code* if and only if the modulo-2 sum of two codewords produces another codeword.

Because a (n,k) linear code C is a k-dimensional subspace of V_n (all the binary n-tuples), it is possible to find k linearly independent codewords g_i, in C, that every codeword in C is a linear combination of these k codewords. Then, we put these k linearly independent codewords in a matrix $G_{k\times n}$. So, if u is a message to be encoded, the corresponding codeword can be given as

$$v = u \cdot G \tag{3.20}$$

The rows of the matrix G can generate the (n,k) linear code C and then it is called a *generator* matrix of C.

A codeword has a certain structure which is called *systematic*. It is divided in two parts. The message part which has k digits is the right part. The remaining $n-k$ digits (left part) is called redundancy. The left hand side digits are also called $parity-check$ digits. A linear block code with this structure is referred as a linear systematic block code. Such a block code can completely specified by a $k\times n$ matrix $G = [P \mid I_k]$, where I_k is the $k \times k$ identity matrix.

In section 3.5 mentioned the $(n-k)\times n$ matrix H where each element of G is orthogonal to the element of H. So, we can define that in a block code with generator matrix $G_{k\times n}$, there is the matrix $H_{(n-k)\times n}$ that

$$G \cdot H^T = 0 \tag{3.21}$$

As a result, every codeword v, which is generated by matrix G, must verify the equation

$$v \cdot H^T = 0 \tag{3.22}$$

, because $v \cdot H^T = (u \cdot G) \cdot H^T = u \cdot (G \cdot H^T) = 0 \cdot u = 0$. The code C is said to be the null space of H. This matrix is called a $parity-check$ matrix of the code. The systematic form of the $parity-check$ matrix is

$$H = [I_{n-k} \mid P^T] \tag{3.23}$$

3.7 Syndrome and Error Detection

Consider an (n,k) linear code with generator matrix G and parity-check matrix H. Let v be a codeword that was transmitted over a noisy channel. Let r be the

received vector at the output of the channel. Because of the channel's noise, r may be different from v. The vector

$$e = r + v$$

is an n-tuple, where $e_i = 1$ if $r_i \neq v_i$ and $e_i = 0$ if $r_i = v_i$, where $0 \leq i \leq n-1$. The vector e is called the *error vector* and displays the positions, where the received vector r differs from the transmitted codeword v. Due to binary operations of the vectors we can declare that

$$r = v + e$$

Unfortunatelly, the receiver does not know if the received codeword has errors. So, he does not know v and e. He knows only r and will try to determine if there are errors and if yes, then must disclose their position in the codeword. In order to determine the existence of errors, the receiver computes

$$s = r \cdot H^T$$

The s is named the *syndrome* and if equals to zero, then the r is a codeword which belongs to the block code. If s not equals to zero, the r is NOT a codeword. Thus, according to the values of syndrome the receiver may safely disclose the sent message.

Though, things may not be always that easy. If an error vector e equals to another codeword of the block code, let's say the codeword l, then

$$s = r \cdot H^T = (e + v) \cdot H^T = (l + v) \cdot H^T = 0$$

because $l + v$ produces another codeword. In this situation we have received an errored mesage which is equal to another message but not the one sent. That kind of errors are not detectable and produce *decoding errors*. These problems are solved introducing a magnitude called *error probability* [17].

3.8 The Minimum Distance of a Block Code

An important parameter of a block code is the *minimum distance*. This parameter determines the $random-error-detecting$ and $random-error-correcting$ capabilities of a code.

If v is a binary codeword, then its *Hamming weight*, $w(v)$, is the number of non-zero digits. For instance, if v=(1 0 0 1 0 1 1) the weight $w(v) = 4$. If u=(0 1 0 0 0 1 1) is another codeword, then the $w(u) = 3$. Their *Hamming distance*, $d(v,u)$, is also 3 and indicates the number of places that have different digits. In this example v and u differ in the first, second and forth position. It is proved that the distance between two codewords is equal to the weight of their sum. So, for the v and u is:

$$d(v,u) = d(1001011, 0100011) = 3$$
$$w(v+u) = w(1001011 + 0100011) = 3$$

which implies

$$d(v,u) = w(v+u)$$

If we introduce another codeword x, then it is:

$$d(v,u) + d(u,x) \geq w(v+u)$$

It is true that for two distinct v and u, in a linear block code, the minimum distance d_{min} ($d_{min} = min(d(v,u) : v \neq u)$) is equal to the minimum weight w_{min} of its nonzero codewords and vice versa. So, for a linear block code, determining the minimum distance of the code is like determining its minimum weigh.

3.9 Error-Detecting and Error-Correcting Capabilities of a Block Code

When a codeword v is transmitted over a noisy channel, an error pattern of l errors will result a received vector r that differs from the transmitted codeword v in l places, which means that $d(v,r) = l$. If the minimum distance of a block code C is d_{min}, then any two different codewords of C differ at least in d_{min} positions. For that code, any error vector e with weight $w(e) \leq d_{min} - 1$ will not produce another codeword that belongs to the block code C . Then, the receiver is sure that he has a modified (errored) codeword named r in less or equal to $d_{min} - 1$ positions and then computes its syndrome. Thus, block code with minimum distance d_{min} is capable of detecting all the error patterns with less or equal to $d_{min} - 1$ errors.

As mentioned in section 3.7, there is the possibility to be transmitted a codeword that its syndrome is zero. In that case it might be transmitted a codeword without any errors or the error vector modified the original codeword to another codeword. There are $2^k - 1$ codewords that belongs to the block code. The 2^k are the possible messages that can be produced by the block code. These are tranfrormed to 2^n codewords through the encoding procedure, minus the zero vector. So, if we subtract the 2^k possible messages from the 2^n possible codewords, $2^n - 2^k$, we get all the detectable error patterns that may create a noisy channel. Based on certain block code algorithms, if n is very large, then 2^k is pretty pretty smaller than 2^n, which means that $2^n - 2^k \simeq 2^n$. Thus, most of the codewords having errors are detected from the decoder because the message has converted to a valid codeword.

CHAPTER 4

The McEliece Cryptosystem

4.1 Introduction

Robert J. McEliece [7], born in 1942, was a mathematician and engineering professor at Caltech and in 1978 proposed the first public key encryption scheme, which is based on algebraic error-correcting codes. He was the 2004 recipient of Claude E. Shanon Award and the 2009 recipient of the IEEE Alexander Graham Bell Medal.

McEliece system belongs to a narrow class of code-based primitives that have impressive resistance against a variety of cryptanalytic attacks and remain unbroken up to now. It recently regained attention with the advent of the emerged field of post-quantum cryptography, as one of the future quantum-resistant standards for public-key encryption. While extremely fast in encryption and reasonably fast in decryption, the McEliece cryptosystem and its variants suffer from having very large key sizes and low transmission rate. In its original formulation, it is built on binary Goppa codes [8] of length $n = 1024$ and dimension $k = 524$ that are able to correct $t = 50$ errors.

As mentioned, a drawback [10] of the McEliece cryptosystem is the comparably large key size in order to hide the well-structured and efficiently decodable Goppa code in the public key, in which is hidden the the generator matrix G of the code. Therefore, several efforts have taken placed in order to reduce the McEliece Cryptosystem key size [11].These efforts have not been possible without incurring into serious security flaws. Though, apart from the key size, the McEliece encryption scheme has many strong features. First, the security reductions are tight, as well as the system is very fast and extremely efficient from an algorithmic complexity perspective.

In a overview, constructing the McEliece cryptosystem is like contructing the *secret* and *public* key. The *secret* key is consisted of a random binary Goppa code Γ, over the binary Galois Field, GF(2). The length is n and the dimension is k and the capability of correcting errors is w (or denoted as t) .

The generator matrix $G_{k\times n}$, the random permutation matrix $P_{n\times n}$ and the non-singular matrix $S_{k\times k}$ are necessary, random created, matrices and kept hidden as

part of the *sectrec* key. The *public* key is nothing else than the multiplication of these matrices, which is

$$G'_{k\times n} = SGP \tag{4.1}$$

4.2 Goppa Code

A classic Goppa code [12] is a linear, error-correcting code that can be used to encrypt and decrypt a message. Let recall an extension of a Galois Field GF(p^m), as mentioned in 3.2, where p is a *prime* number and m is a positive integer.

A polynomial $g(x)$, which is called Goppa polynomial, will be the

$$g(x) = g_0 + g_1x + ... + g_tx^t = \sum_{i=0}^{t} g_ix^i \tag{4.2}$$

with each $g_i \in$ GF(p^m), $t \geq 1$. Recall that $p^m = q$ and that t stands for the error correcting capability of a block code. Let, also, L be a finite subset of the extension field GF(p^m), with

$$L = \{a_1, a_2, ...a_n\} \subseteq GF(p^m) : g(a_i) \neq 0, \forall a_i \in L \tag{4.3}$$

Given a codeword $c = (c1, .., c_n)$ over GF(q), we can have the function

$$R_c = \sum_{i=1}^{n} \frac{c_i}{x - a_i} \tag{4.4}$$

where $\frac{1}{x-a_i}$ is the unique polynomial with $x - a_i \cdot \frac{1}{x-a_i} \equiv 1 (mod\, g(x))$, with a degree less than or equal to $t-1$. Then, a Goppa code $\Gamma(L, g(x))$ is made up for all code vectors c such that $R_c(x) \equiv 0 (mod\, g(x))$. This means that the polynomial $g(x)$ divides the $R_c(x)$.

4.2.1 Parameters of a Goppa Code

Considering chapters 3.6 and 3.7 someone can use the notation $[n, k, d]$ to describe Goppa code. These parameters are :

(i) n: The *length* of the code

(ii) k: The *dimension* of the code

(iii) d: The *Hamming* distance of the code

The parameter n depends of the subset L of the GF(2) extension mentioned in the previous topic (see 4.3).The parameter k is related to n such that:

$$k \geqslant n - mt$$

and the d parameter is related such that:

$$d \geqslant t + 1$$

where m is the positive integer superscript of the extension of the Galois field.

4.2.2 Binary Goppa Code

A classic Goppa code is refered when the number p is a prime number and m is a positive integer. When p equals to 2, then a special kind of Goppa code is created, the Binary (modulo-2) Goppa code [12]. So, such a code $\Gamma(L, g(x))$ should use a polynomial $g(x)$ over the GF(2^m). The degree of the polynomial will be t. Recall from 4.1 that t is the capability of error correction of the code. The sake of the classic McEliece cryptosystem is based on the irreducible binary Goppa code.That code is irreducible when the $g(x)$ is irreducible which means that it cannot be factored in to two non-constant polynomials. In other words, a polynomial over GF(2^m) is irreducible if it is not divisible by any polynomial with lesser degree, over the same field. For example, let introduce the polynomial, over GF(2^m):

$$p_1(x) = 1 + x + x^3$$

Any other non-constant polynomial with lesser degree should contain the x and/or x^2. Such polynomials does not completely devides the $p_1(x)$ and so, it is irreducible.

In a Binary Goppa Code the parameter of distance d is modified in contrast of the simple Goppa Code and should be greater than $2t + 1$. So, the parameters become

$$[n, \geqslant n - mt, \geqslant 2t + 1] \tag{4.5}$$

4.2.3 Parity Check and Generator Matrix

The parity check matrix of a Goppa code is used in order to decode a message. It is usually appeared as H and defined as the matrix where

$$cH^T = 0$$

for all the code vectors $c \in$ block code $\Gamma(L, g(x))$. In detail, the parity check matrix H is consisted of three matrices, let's say X, Y, Z, where

$$H = XYZ$$

and

$$X = \begin{pmatrix} g_t & 0 & \cdots & 0 \\ g_{t-1} & g_t & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ g_1 & g_2 & \cdots & g_t \end{pmatrix}$$

$$Y = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ \alpha_1 & \alpha_2 & \cdots & \alpha_n \\ \vdots & \vdots & \ddots & \vdots \\ \alpha_1^{t-1} & \alpha_2^{t-1} & \cdots & \alpha_n^{t-1} \end{pmatrix}$$

$$Z = \begin{pmatrix} \frac{1}{g(\alpha_1)} & 0 & \cdots & 0 \\ 0 & \frac{1}{g(\alpha_2)} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{1}{g(\alpha_n)} \end{pmatrix}$$

with $g(a_i) \neq 0 \forall a_i \in L$, as mentioned in 4.3.

If the parity check matrix is equivalent to

$$H = [P \mid I_{n-k}] \tag{4.6}$$

then the corresponding generator matrix would be

$$G = [I_k \mid -P^T] \tag{4.7}$$

From 4.7 and 4.6, it is

$$GH^T = P - P = 0$$

and the equation 3.21 is verified (I is denoted the Identity Matrix).

4.3 Construction of a McEliece Cryptosystem

4.3.1 Preliminaries

The McEliece cryptosystem is a Public Key Cryptosystem, which means that it uses a public key and a private key in order to encrypt and decrypt a message. Reconsider the paradigm in Sec. 2 where Alice and Bob would like to communicate safely. Now, Eve, an evil person, tries to acquire information from that communication.

In order Bob to avoid Eve's evil activities, he constructs his public key by selecting a binary Goppa irreducible polynomial $g(x)$ of degree t. To do so, he should utilize the python function

$$get\_irreducible(t\_val)$$

The function is listed in appendix A.2.1 and is enclosed in the major module *McElieceKeygen.py*.

Then, he computes the parity check matrix H. To do that he uses the function

$$get\_H(m\_val, t\_val, k\_vec, g\_vec), (A.2.2)$$

It is obvious that in the first lines is computed the $g(a)$ and the $\frac{1}{g(a)}$, needed to construct H. The rational of this procedure is analyzed in Ch. 4.2.3. Also, the parameter k_vec includes all the factors of the polynomial $K(x)$, an irreducible polynomial that represents all the elements of the selected Galois field, whereas $g(x)$ represents the elements of the subspace L (see Ch. 4.3). Then, using H, he computes the generator matrix G (see Appx A.2.3) of that Goppa code, combining the Eq. 4.6 and 4.7 .

4.3.2 Public and Secret Key

In order to generate both public and secret key Bob chooses a random invertible matrix S, a random permutation matrix P and uses them to compute the G', which is represented as G_pub in Appendix A.2.4. At the same Appendix, in lines 33 and 38, there are the functions that generates the matrices S and P. In line 42 is computed the G_pub and the H_pub. The latter is the public version of the mystic parity check matrix. That version of H is said to be known for the sake of some attack algorithms, hence

$$H\_pub = HP$$

Then, he publishes his public key which consists (G', t). His private key is consisted of (S, P, G). He should not publish the private at all costs. Afterwards, via the function *writeKeys*, the *McElieceKeygen* python module writes all the keys to respective files in a manner that the McEliece system may use to encrypt, decrypt or even encode-decode plain text. Breaking down that function (see Appx A.2.5), the lines

```
1 with gzip.open(filename +' .'pub, ''wb) as f:
2 f.write(pickle.dumps([G_matrix, t_val]))
```

extract the public key and the lines

```
1 with gzip.open(filename +' .'priv, ''wb) as f:
2 f.write(pickle.dumps([S_matrix, H_matrix, P_matrix, t_val]))
```

the secret key. Additionally the function *writeKeys* extracts the H_pub and G (G is used only for internal checks).

The names of the extracted files follow a certain pattern. For instance, *mt.pub* is the public key, where m and t are the necessary and well known values that construct the block code. Thus, if the Galois field extension integer $m = 4$ and the capability of error correcting $t = 3$, then after executing the program we will find the public key file as 43.*pub*.

Code Listing 4.1: McEliece Key Generation

```
create_keys_cword.sh 4 3
```

4.3.3 Encryption

An abstract view of encrypting a message m is shown in Appx A.1.1. In a more technical point of view, Alice would follow the sequense:

(i) write her message m,

(ii) convert it into a binary,

(iii) encrypts the encoded message using Bob's public key (mG'),

(iv) generates a random error vector e with weight of t, or less, and add it to the encrypted message ($mG' + e$)

(v) sends the cipher text $c = mG' + e$ to Bob

In Appx A.2.6 there is an implementation of the aforementioned sequence . The whole message is separated to blocks and each block is encrypted separately.

For example, in order to encrypt a message, over the field $m = 9$ and $t = 9$, one should follow the following procedures:

(i) Write the message (for example 'hello world').

(ii) Convert the plain text message to binary.

To encode, we use the python module *MatrixCodec.py* in a linux terminal as follows:

Code Listing 4.2: McEliece Encode Raw Text

```
python MatrixCodec.py -e hello world -par 99.parity
```

The module *MatrixCodec.py* takes a raw text as input and when the switch $-e$ is parsed it performs the encoding process where converts the raw text to binary vectors of k length. The switch $-par$ is used in order the encoder reveal the parameter k which is hidden in the parity check matrix H. That parameter is mandatory for the encoding procedure because the whole message must be chopped to $k - length$ parts. For more investigation the manual page is :

```
usage: MatrixCodec.py [-h] [-o O] [-e E] [-d D] [-k K] [-v] [-
    par PAR]

optional arguments:
  -h, --help  show this help message and exit
  -o O        File to store output
  -e E        String to be encoded into matrices
  -d D        File with matrices to be decoded to ASCII
  -k K        Length of each matrix
  -v          Enable verbose mode
  -par PAR    Parity matrix
```

(iii) Encrypt the binary message.

Then, takes over the module *McEliece.py* which executes the encryption function A.2.6 when we parse, simultaneously, the switches *−e* <*binary message*> , *−pub* <*public key*>, *−o* <*codeword*>. In command live view this could be

Code Listing 4.3: McEliece Encrypt Binary Message

```
python McEliece.py -e 99.binMsg -pub 99.pub -o codeword
```

Programmingly, the generated codeword in this step includes also the addition of a random error vector. For more details see the manual:

```
usage: McEliece.py [-h] [-v] [-vv] [-g] [-m M] [-t T] [-o O] [-e E]
    [-d D]
                   [-pub PUB] [-priv PRIV] [-f] [-x X] [-par PAR] [-
                      cw CW]
                   [-er ER] [-pErr PERR] [-l L] [-q Q]

optional arguments:
  -h, --help  show this help message and exit
  -v          Enable verbose mode
  -vv         Enable very verbose mode
  -g          Generate key pairs
  -m M        Generate key pairs
  -t T        Generate key pairs
  -o O        Output file (always needed)
  -e E        File with data in matrices to encrypt
  -d D        File with data in matrices to decrypt
  -pub PUB    Key to encrypt with
  -priv PRIV  Key to decrypt with
  -f          Encrypt without errors in ciphertext
  -x X        Encoded msg without errors
  -par PAR    Parity matrix
  -cw CW      Codeword
  -er ER      error vector
  -pErr PERR  errors in section A(stern)
  -l L        size of dimension l(stern)
  -q Q        errors of dimension l(ballcoll)
```

4.3.4 Decryption

Decryption is slightly more complicated. Firstly, the receiver of the codeword c needs to remove the error vector e and then decode the ciphertext. Afterwards, he can utilize the hidden structure of the Goppa code, that includes an efficient

decoding method, and also use the private key in order to retrieve mS. So, once e is known, the message (denoted as m) can be easily revealed following the Algorithm A.1.2.

Decryption is a highly demanding procedure, in respect of sources, because it follows the syndrome decoding method. Hence, the decoder must compute the possible error vectors, as it knows the value t. Thus, the more errors enclosed in the cipher the more syndromes must be calculated.

From development perspective, in order the python module *McEliece.py* switched to decryption mode, using the *decrypt* python function (see Appx A.2.7), one should parse:

(i) File with data in matrices to decrypt $-d$

(ii) Key to decrypt with $-priv$

(iii) Output file $-o$

Applying these parameters to the *McEliece.py* we can reveal the plain text message by typing :

(i)
```
McEliece.py -d 79.codeword -priv 99.priv -o 99.plain
```

(ii)
```
MatrixCodec.py -d 99.plain
```

Eve constantly wishes to had Bob's private key in order to decrypt Alice's message. Fortunately, she cannot access his hidden files. It is the fundamental principle of the asymmetric key encryption method to store locally the private key. Nevertheless, there have being developed several methods that may attack the public key cryptography, in respect of confidentiality or data integrity.

4.4 Basis of Security of the McEliece Cryptosystem

One of the basic principles of cryptography is that the security of a system must not depend on keeping secret the algorithm but keeping secret only the private key. This principle is known as Kerckhoff's Principle [13]. So, let's assume that Eve knows the cryptosystem used and then she may try to decrypt Alice's message, without knowing the private key. She would have a difficult time. For the McEleice cryptosystem that difficulty happens because she needs to reveal matrix G from the matrix G'. But, the matrix G' is not invertible and so, she have to know the inverse of the chosen random matrix S which was not published. Eve also, does not know what the matrix P and so cannot find c' and later find m', which reveals the hidden message m, according to the decryption algorithm.

Analysing the $G' = SGP$, the multiplication SG scrambles the message into another matrix. Then, when we multiply the result by the permutation matrix P,

we scrable more the matrix by randomizing the order of the columns. This is done in order to make the resulting matrix look random and ensure the difficulty of obtaining the decoding method from the encoding matrix. Because of this scrambling of the matrices, Eve cannot decipher the private key from the public key.

Basically, in order to keep this cryptosystem secure, it must be very hard to decode c' and find m'. In order to do this, the Goppa code is selected to be as large as possible. For example, in the original McEliece cryptosystem published in 1978 [7], the author suggested the use of a $[n, k] = [1024, 524]$ Goppa code, i.e. a Goppa code of length 1024 and dimension 524. Though, repeatedly sending the same message with the same G' will make the cryptosystem more vulnerable to attacks.

4.5 Hardware Implementation

4.5.1 Embedded Devices

For many embedded systems such as prepaid phones or micropayment systems, the short life cycle or comparably low value of the enclosed product often does not demand a very long-term security. Hence, mid-term security parameters for public-key cryptosystems providing a comparable security to 64-80 key bits of symmetric ciphers are often regarded sufficient (and help reducing system costs). The current implementations are designed for security parameters that correspond to an 80 bit key size of a symmetric cipher. A second important design requirement is the processing and storage of the private key solely on-chip. So all secrets are optimally never used outside the device.

4.5.2 General

There are some assumptions, requirements and restrictions when is needed to implement the original McEliece cryptosystem on small embedded systems. Target platforms, [33] are 8-bit AVR microprocessors as well as low-cost Xilinx Spartan-3AN FPGAs. Some devices of these platforms come with large integrated Flash-RAMs (e.g., 192 kByte and 2,112 kByte for an AVR ATxMega192 and Spartan-3AN XC3S1400AN, respectively).

Analyzing McEliece encryption and decryption algorithms, the following arithmetic components are required supporting computations in $GF(2^m)$: a multiplier, a squaring unit (calculation of square roots) and an inverter. Furthermore, a binary matrix multiplier for encryption and a permutation element are needed, too. Many arithmetic operations in McEliece can be replaced by table lookups to significantly accelerate computations at the cost of additional memory. For both implementations the primary goal is area and memory efficiency to fit the large keys and required lookup-tables into the limited on-chip memories of our embedded target platforms.

4.5.3 Key Size Reduction

To make McEliece-based cryptosystems more practical (i.e., to reduce the key size), there is an ongoing research to replace the code with one that can be represented in a more compact way. Using an approach in which the support of the code is the set of all elements in $GF(2^m)$. While the matrices S, P are totally random, the public key $G' = S \times G \times P$ is a random $n \times k$ G matrix. Then, since P is a permutation matrix, with only a single 1 in each row and column, it is more efficient to store only the positions of the ones, resulting to an array with $n \cdot m$ bits.

Another trick to reduce the public key size is to convert G' to systematic form $\begin{pmatrix} I_k & Q \end{pmatrix}$, where I_k is the $k \times k$ identity matrix. Then, only the $k \times (n-k)$ matrix Q must be published.

4.6 Lightweight Co-Processor

4.6.1 General

The conventional McEliece cryptosystem uses the binary Goppa code, which has good code rate and error correction capability. However, comparing with other binary codes, the generating and decoding (error-correction) of such codes have relatively high complexity since they involve intensive computations over finite fields, including modulo polynomial operations. In addition, the key size of McEliece systems is usually large. For a binary Goppa code with an $k \times n$ generator matrix, the key size is $k \cdot n$ with k, n in thousands of bits, which is one of the drawbacks of the system, as mentioned before (see Ch. 4.1).

Therefore, to address these issues Bu et al [32] proposed a new variant of McEliece cryptosystem and its encryption-decryption co-processor. The proposed system is based on the generalized non-binary Orthogonal Latin Square Code (OLSC) [34], which is known for its simple encoding and decoding algorithms, leading to potential efficient hardware implementation. Additionally, the non-binary OLSC is able to work with non-binary messages through binary matrices. In other words, a long message is able to be processed by relatively small matrices, which reduces the key size.

4.6.2 Analysis

The McEliece based public key encryption cryptosystem co-processor has three modules: the Key Generation, Encryption and Decryption, shown in Fig. 4.1. Of these three modules, the decryption unit has the highest complexity and consumes the most hardware resources, especially at the decoding stage. Therefore, the current implementation primarily targets to the efficiency of the decode sub-module.

Decoding algorithm is a decoding procedure of OLSC codes, which consists mostly of binary linear operations. Thus, it can be carried out fast and efficient in hardware. Bu manages to generate the binary OLSC to non-binary codes, while

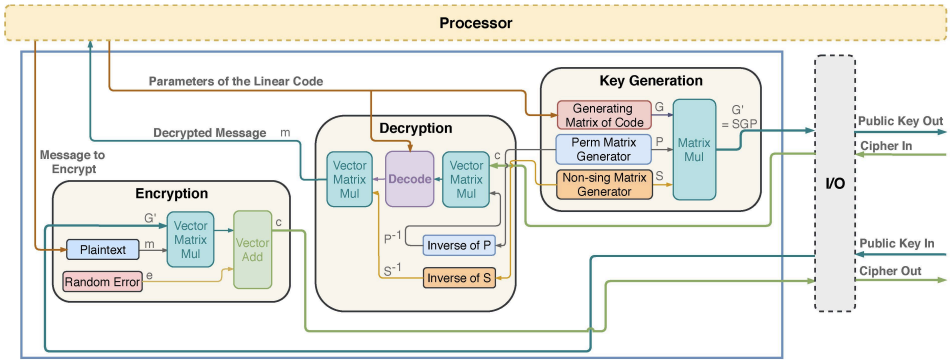

Figure 4.1: McEliece Cryptosystem Co-Processor Architecture

still maintaining low decoding complexity. By replacing the binary Goppa code with the non-binary OLSC, the decoding stage only requires less resources. This feature enables much faster decryption time than the original binary Goppa code scheme. (see Tab. 4.1).

Table 4.1: Complexity Analysis

	Finite Field Ops	Latency (cycles)
Binary Goppa Code-based	$O(n^2)$	$O(n)$
OLSC-based	0	$O(1)$

CHAPTER 5

Attacking McEliece Cryptosystem

5.1 Brute Force Perspective

In cryptology, there are various types of attacks. A brute force attack is when the enemy tries every possible key and determines which key results a meaningful message. For such an attack, the longer the length of a key, the longer it will take to try every possible key. A direct, or per-message, attack is an attack that tries to decode a given message but does not necessarily solve the entire cryptosystem. A structural attack occurs when an eavesdropper would try to recover the structure, or at least part of the structure, of the original message utilizing the public key.

As mentioned before, selecting large numbers for n and t, a brute force attack approach seems infeasible because there are a lot possibilities for constructing G from a numerous Goppa irreducible polynomials. Including the random P and S matrices which scrambles the generator matrix, the complexity is increased dramatically. In particular, suppose someone chooses length $n = 1024 = 2^{10}$ and $t = 50$. Thus, there are 10^{149} Goppa polynomials and an enormous number to choose P and S. Considering the Eq. 4.5, the dimension of the code will be $k = 1024 - 50 \cdot 10 = 524$. Hence, a brute force attack based on comparing the received codeword r versus the real codeword c, belonging at the block code, would have a work factor 2^{524} [7]. Finally, the general decoding problem for linear codes is an NP-Complete problem and such problems have not efficient solving time.

At his thesis 'A Coding-Theoretic Approach to Cryptanalysis', Alexander Meurer mentions that the brute force attack (BF) in general, costs

$$2^{BF(R)n+o(n)} \tag{5.1}$$

steps, where $R = \frac{k}{n}$ is the rate of the code and

$$BF(R) = \begin{cases} R & ,0 \leq R \leq \frac{1}{2} \\ 1-R & ,\frac{1}{2} < R \leq 1 \end{cases}$$

In his assumptions on delivering that cost (Eq. 5.1) he ignores the polynomial complexity when calculating some trivial amounts, for instance the syndrome of a

candidate error vector. So, for the $n = 1024$ and $k = 524$ block code, the equation 5.1 is equal to $2^{500+o(1024)}$, almost similar to what McEliece reported in his initial paper.

5.2 Quantum Perspective

Assume that large quantum computers are built and that they function as smoothly as one could possibly hope. Shor's algorithm and its generalizations will then completely break RSA, DSA, ECDSA and many other popular cryptographic systems. For example, a quantum computer will find an RSA user's secret key at essentially the same speed that the user can apply the key.

That algorithm is not the only application of quantum computers. A quantum searching algorithm, introduced by Grover in [20] and [21], finds a 256-bit AES key in only about 2^{128} quantum operations given a few known plaintexts encrypted under that key. Users who want to push the attacker's cost significantly higher than 2^{128}, the original motivation for 256-bit AES will need a cipher with significantly more than a 256-bit key.

According to [22], there is no reason to think that the doubled key size for a secret key cipher will be matched by a doubled hash function output length, a doubled key size for McEliece, etc. Quantum computers have only a small impact on the McEliece public key system, reducing the attacker's decoding cost from 2^{140} to 2^{133} for a code of length 4096 and dimension $4096 - 45 \cdot 12 = 3556$.

Grover's algorithm takes only square root time compared to a brute force key search but this does not mean that it takes less time than the more sophisticated algorithms used to find hash collisions, McEliece error vectors, etc. Sometimes Grover's idea can be used to speed up the more sophisticated algorithms but understanding the extent of the speedup requires careful analysis. Such a sophisticated area of algorithms is called Information-Set-Decoding where is faster than the brute force 'Structural Attacks' and aims to reveal the structure of the Goppa code. The structure is the secret generator matrix G.

5.3 Information Set Decoding (ISD) Perspective

Basic rationale of the Information Set Decoding against the structure of a simple cryptosystem based on generator matrix G is: choose a uniform random k-size subset $S \subseteq \{1, 2, ..., n\}$ and consider the natural projection $GF_2^n \rightarrow GF_2^S$ that extracts the coordinates indexed by S, discarding all other coordinates. Assume that the following two events occur simultaneously:

(i) The error vector e projects to $0 \in GF_2^S$ (means that the error bits are in the subspace $N - S$).

(ii) The composition $GF_2^k \xrightarrow{G} GF_2^n \rightarrow GF_2^S$ is invertible. Thus, the columns of G indexed by S form an invertible $k \times k$ submatrix.

The above implies the Gaussian elimination needed to Obtain message m by applying the inverse to the projection of $mG + e$, and obtain e by subtracting mG from $mG + e$. If this fails, i.e. if the composition is not invertible or if the resulting e does not have Hamming weight t, then go back to the beginning and try another set S.

According to [28], the first event occurs for exactly $\binom{n-t}{n}$ combinations out of $\binom{n}{k}$ choices of the subset S. So, it occurs in average iterations

$$\frac{\binom{n-t}{k}}{\binom{n}{k}} \tag{5.2}$$

When $t \simeq \frac{(1-R)n}{logn}$ then the Eq. 5.2 $\in c^z, z = \frac{(1+o(1))n}{logn}$.

Furthermore, for any particular S, the second event occurs for approximately 29% of all matrices G because approximately 29% of all $k \times k$ matrices are invertible. So, for a chosen generator matrix G the second event occurs for 29% of choises of S. Combining the two events we get $0.29\binom{n-t}{k}$ invertible G out of $\binom{n}{k}$ choises of S. This assumption appears to be correct for McEliece public keys. Additionally, conscerning the complexity of the brute force attack in Ch. 5.1 for the [1024,524] block code, we see that the brute force costs 2^{524} and the simple ISD method costs 1.42^{300} (see Eq. 5.2).

More advanced forms of information-set decoding complicate each iteration but decrease the number of iterations required, for example by allowing e to have a few bits set within S and searching for those bits. There are also many techniques to reduce the cost of finding appropriate sets S, computing inverses, checking whether m is correct.

5.4 Information Set Decoding Algorithms

Information Set Decoding can be seen as a class of generic decoding algorithms for the CSD problem for random linear codes or equivalently for codes without any particular structure. To apply that kind of algorithms we assume that the wanted error vector has weight exactly t and we do know that value (it is part of the public key). This knowledge of t merely allows to define the brute force search space as the set $W_{n,t}$ and to restrict the search to particular error patterns that occur with a certain probability (which can be easily computed). Exploiting the vector space structure essentially corresponds to applying elementary transformations to parity check matrix H. In particular, permuting the columns of H allows one to randomly shuffle the error positions of the error vector e. Thus, ISD simply provides a clever way to reduce the brute-force search space by linear algebra.

5.4.1 Prange's Algorithm

In 1962, Eugenie Prange introduced the first ISD algorithm [24], which is often called 'plain information set decoding', and thus created the basis for all the subsequent improvements. Plain ISD simply repeats permutations of the error vector coordinates until all errors finally occur in the last $n-k$ positions only. So, the error vector e will be separated like:

$$e = \begin{pmatrix} k & n-k \end{pmatrix}$$

The main idea of plain ISD is fairly simple:

(i) Compute the standard form $\tilde{H}$ of a randomly column-permuted public version of parity check matrix H_{pub}, for simplicity notated as H. The public version is the one multiplied with the permutation matrix P, chosen during the public key construction.

(ii) Hope that no errors occur in the first k coordinates (in that case the syndrome $\tilde{s}$ will reveal the permuted version $\tilde{e}$ of e)

Repeatedly computing such randomise version of the public parity check matrix will then form the version of the error vector with the errors in the last $n-k$ positions.

In particular, firstly we compute the syndrome $s = cH^T$, where c is the received codeword. Then, in order to randomise the version of H, we multiply it with a random permutation $n \times n$ matrix P. Applying Gauss Elimination we transform HP to the systematic form $(B\ I_{n-k})$. The $k \times k$ non-singular matrix Q that is capable of converting to systematic form appears in the equation as

$$\tilde{H} = QHP = (B\ I_{n-k}) \tag{5.3}$$

In other words, the Gauss Elimination is successful if and only if there is an invertible matrix Q that

$$HP = (HP_L\ HP_R) \Leftrightarrow QHP = Q(HP_L\ HP_R) = (B\ I_{n-k})$$

which means that the HP_R must be invertible, thus capable to produce the identity matrix. If not, the algorithm must select another permutation matrix P.

Using the matrix Q we can compute $\tilde{s} = sQ^T$ which is actually the $\tilde{e}$ from the coordinates $k+1$ to n . To fully represent the $\tilde{e}$ we use the form $\tilde{e} = (0\ \tilde{s})$. As mentioned before, a good P is the one that occurs the non zero coordinates to the last $n-k$. If the weight of the $\tilde{e}$ (or similarly the $\tilde{s}$) is equal to t, then we can to reveal the error vector as $e = \tilde{e}P^T$. If not, we select another permutation matrix P.

A brief pseudo code implementation of Prange's algorithm is pinned in the Appx. A.1.3, where an implementation in python is in Appx. A.3.1.

In praxis, we executed the *prange.py* module for several values of Galois Fields, in other words for several rate values, in value space $(0,1)$. The graph in Fig. 5.1 shows the result of running multiple times the Prange (Plain-ISD) algorithm and the corresponding time. Analysing the results, regrading the message bit rate R, we may deduct that in:

(i) Low rate: We encounter a lot of errors t, hence small message length k ($k = n - mt$). Thus, $n - k \simeq n$. So, it is easier to assume that the errors exist in the $n-k$ coordinates which is almost the whole codeword. inevitably leads to smaller running time.

(ii) High rate: We encounter few errors t and so bigger message length value k. But, with few errors is easier to put them at the $n-k$ coordinates, which also leads to smaller running time.

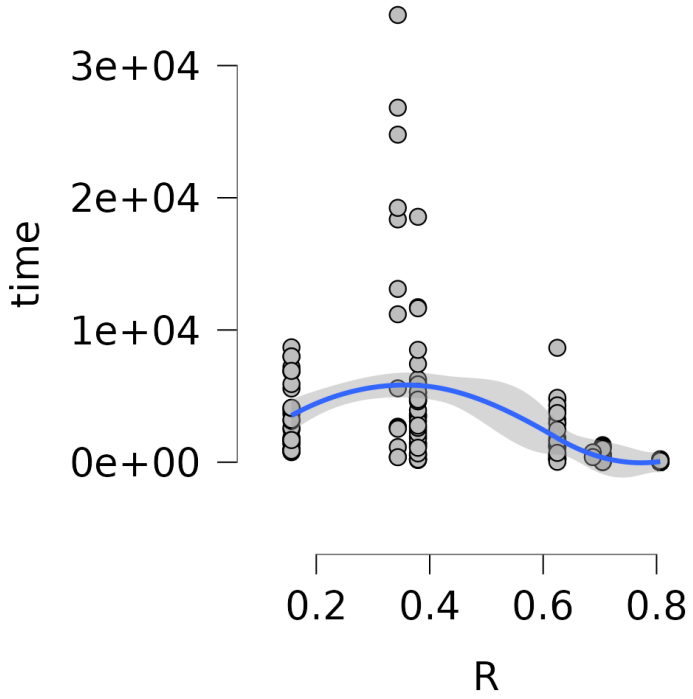

Figure 5.1: Plain-ISD (Prange) execution time

5.4.1.1 Complexity

The complexity of the Plain-ISD algorithm relies in two factors. The first is the Gauss Elimination, which requires $T_{GE}(n-k)$ operations. This costs

$$T_{GE}(x) \simeq \frac{1}{2}x^2 n \tag{5.4}$$

On the other hand, the complexity depends on the probability of founding a good permutation, where we need to have all the errors at the $n-k$ coordinates of the code word. This is quantified as

$$Pr_{Pl-ISD} = \frac{\binom{n-t}{k}}{\binom{n}{k}} \tag{5.5}$$

Combining the 5.4 and 5.5 we may say that the total number of operations are

$$N_{Pl-ISD}(n,k,t) = \frac{T_{GE}(n-k)}{Prob_{Pl-ISD}}$$

To investigate more the factor of the probability, we select two rate points, the $R = 0.2$ and $R = 0.4$, where time execution seems to be bigger. At these rates the number of operations N are to be higher because the probability tends to be small. For example, the nominal probability to get a good permutation for the rates mentioned in this paragraph is:

$$Pr_{Pl-ISD}(R = 0.2) = 0.048 \tag{5.6}$$

$$Pr_{Pl-ISD}(R = 0.4) = 0.193 \tag{5.7}$$

Let us assume that we run an algorithm n times (n here stands for the statistic sample). Let, also, denote as *attempts* the number of the internal loops that Plain-ISD do trying to reach a good permutation matrix P and aiming to find the mystic error vector. Then, we can declare the experimental probability of finding a good permutation as the

$$Pr_{exp.} = \frac{1}{attempts} \tag{5.8}$$

For the the sake of our experiment, we choose the $R = 0.2$ and run the *prange.py* module for $n = 41$ times (set with the switch -c).

Code Listing 5.1: Prange Execution

```
python prange.py -c 41 -m <the GF extension integer> -t <number of
    errors>
```

For more details about the usage of the prange.py , one may run :

```
prange.py -h

usage: prange.py [-h] [-c C] [-t T] [-m M]

optional arguments:
  -h, --help  show this help message and exit
  -c C        Times to repeat the alg
  -t T        num of errors
  -m M        m in GF
```

So, if we go through the python implementation of the ISD algorithms we may notice that each output is the number of attempts needed (or else the number of internal loops) to finding a good permutation, named as *attempts*. Using the Eq. 5.8 we compute the experimental value for the probability of having a good permutation.

For this experiment, the probability mean value that measured is 0.073, which is almost similar to the nominal value $Pr_{Pl-ISD}(R = 0.2) = 0.048$.

Table 5.1: Experim. Plain-ISD for R=0.2

Variable	n	Mean	Std. deviation
pr.	41	0.073	0.176

We may also see from the Fig. 5.2 that the experimental probability, $Pr_{exp.}$, is 66% likely to appear with greater value than the experimental mean (also greater than the nominal value).

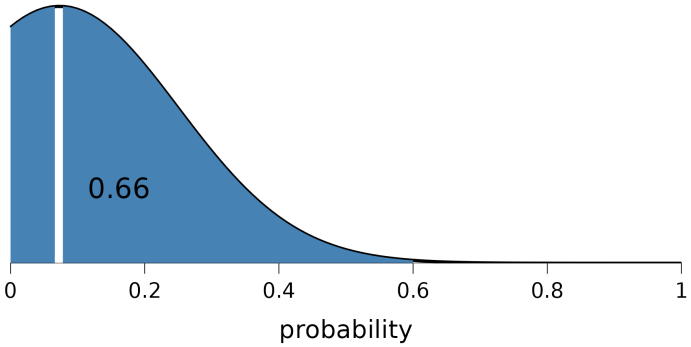

Figure 5.2: Experim. Plain-ISD for R=0.2

At the same conclusions we arise when we repeat the experiment for $R = 0.4$. Running the *prange.py* for 34 times we get the mean value 0.196, slightly similar to the nominal $Pr_{Pl-ISD}(R = 0.4) = 0.193$ (5.6).

Table 5.2: Plain-ISD for R=0.4

Variable	n	Mean	Variance	Std. deviation
pr.	34	0.196	0.079	0.281

According to Fig. 5.3 , like the previous situation with $R = 0.2$, it is more likely (68%) to get a good permutation with probability greater than the experimental value, which is almost the same as nominal.

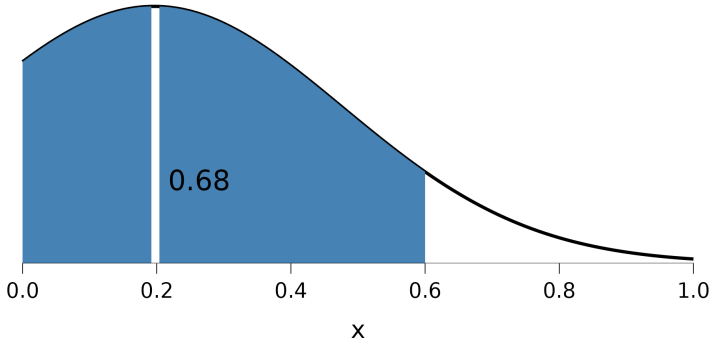

Figure 5.3: Plain-ISD for R=0.4

5.4.2 Stern's Algorithm

In 1989, Jacques Stern [25] introduced a method to attack McEliece cryptosystem without the need to accumulate the errors at the last $n-k$ coordinates. Evolving Prange's proposal, he proposed to separate the n coordinates of the error vector at a fashion

$$\begin{array}{cccc} \leftarrow k/2 \rightarrow & \leftarrow k/2 \rightarrow & \leftarrow l \rightarrow & \leftarrow n-k-l \rightarrow \\ p & p & 0 & t-2p \end{array}$$

where l to be a small number.

Stern admits that one can encounter p errors in each $\frac{k}{2}$ coordinates. At the l part, which is pretty small, he hopes that there are not any errors. Thus, at the last $n-k-l$ coordinates we will have the remaining $t-2p$ errors.

In Alexander's May [27] analysis upon the Stern's alg., the parity check matrix $\tilde{H}$ is sliced into two parts. Beginning from the first l rows (upper part), it is

$$\tilde{H_l} = \begin{pmatrix} U & I_l & 0 \end{pmatrix}$$

He also splits the syndrome $\tilde{s}_{[1\times n-k]}$ into two parts, $\tilde{s_l}$, $\tilde{s_R}$ as:

$$\tilde{s} = \begin{pmatrix} \tilde{s_l} & \tilde{S_R} \end{pmatrix}$$

At the same way he decomposes the $\tilde{e}$ as:

$$\tilde{e} = \begin{pmatrix} \tilde{e_L} & \tilde{e_l} & \tilde{e_R} \end{pmatrix} = \begin{pmatrix} \tilde{e_L} & 0 & \tilde{e_R} \end{pmatrix}$$

, with $|\tilde{e_L}| = k$, $|\tilde{e_l}| = l$ and $|\tilde{e_L}| = n-k-l$.

Then, one can further split the $\tilde{e_L}$ in two parts with $\frac{k}{2}$ dimension each, the $e_{\tilde{L_1}}$ and $e_{\tilde{L_2}}$, with p number of errors each. Analogously splits the submatrix U into A and B. Thus,

$$\begin{aligned} \tilde{e} &= \begin{pmatrix} e_{\tilde{L}1} & e_{\tilde{L}2} & 0 & \tilde{e_R} \end{pmatrix} \\ \tilde{H_l} &= \begin{pmatrix} A & B & I_l & 0 \end{pmatrix} \end{aligned}$$

For all these vectors $e_{\tilde{L_1}}$ and $e_{\tilde{L_2}}$, with $wt(e_{\tilde{L_1}}) = wt(e_{\tilde{L_2}}) = p$, since is fulfilled the equation

$$\tilde{s_l} = \tilde{e_L}U^T + \tilde{e_l}I_l = \tilde{e}_{L_1}A^T + \tilde{e}_{L_2}B^T + 0I = \tilde{e}_{L_1}A^T + \tilde{e}_{L_2}B^T \tag{5.9}$$

then we can denote the right part of the error vector $\tilde{e}$ as

$$\tilde{e_R} = \tilde{s_R} + e_{\tilde{L_1}}C^T + \tilde{e}_{L_2}D^T$$

where

$$\tilde{H}_{n-k-l} = \begin{pmatrix} C & D & 0 & I_{n-k-l} \end{pmatrix}$$

is the remaining lower part (the $n-k-l$ rows) of the sliced parity check matrix. From that point, if the $wt(\tilde{e}) = t$ we can reveal the error vector with the help of the permutation matrix P. The steps of the algorithm is in A.1.4.

5.4.2.1 Complexity

To analyze Stern's algorithm we have to consider both the complexity of each iteration and the probability of success. The complexity of each iteration is dominated by the collision finding step, in the list containing the vectors with length $\frac{k}{2}$. This can be done by a simple sort-and-match technique which is identical to two nested for loops.

To analyze the success probability, we need to compute the probability that a random permutation of the error vector e with weight $wt(e) = t$ has a good weight distribution which means that the $\tilde{e}$ must have certain errors in certain position range, as described above. So, the success probability is expressed, analogously to Prange's rationale, as

$$Pr_{Stern} = \frac{\binom{k/2}{p}^2 \binom{n-k-l}{t-2p}}{\binom{n}{t}}$$

Another factor to be consider when we need to represent the complexity, is the the sum of some major computations needed. Such computations are:

(i) The Gauss Elimination in order to convert the public version of the parity check matrix to systematic form. Denoted as $T_{GE}(n-k)$.

(ii) The complexity of the nested loops when they try to find the collisions between the vectors $e\tilde{}_{L_1}$, $e\tilde{}_{L_2}$ and sets L_1, L_2, respectively. Due to equality of the lists length it can be denoted as $2|L^{k/2}|$.

Considering the above, the overall complexity can be computed as

$$N_{Stern}(n,k,t,p,l) = \frac{C_{Stern}}{Pr_{Stern}} = \frac{T_{GE}(n-k) + 2|L^{k/2}|}{Pr_{Stern}} \tag{5.10}$$

5.4.2.2 Stern vs Stern

It is obvious from Eq. 5.10 that the complexity depends from the parameter p which is the number of errors at the k coordinates of the error vector. Putting hands on, for a block code with $[n,k] = [128,86]$ and selecting: $p = 1$ errors in $k/2$ coordinates, $l = 2$, we run the algorithm as follows:

Code Listing 5.2: Stern Execution, p=1, l=2

```
python stern.py -c 40 -m 7 -t 6 -p 1   -l  2
```

As a result we get experimental probability 0.047, whereas the nominal value is $Pr_{stern(p=1)} = 0.03$. Also, according to further manipulation of the experiment's data, we may deduct that the probability of having a good permutation is slightly greater than the nominal value (see Fig. 5.4).

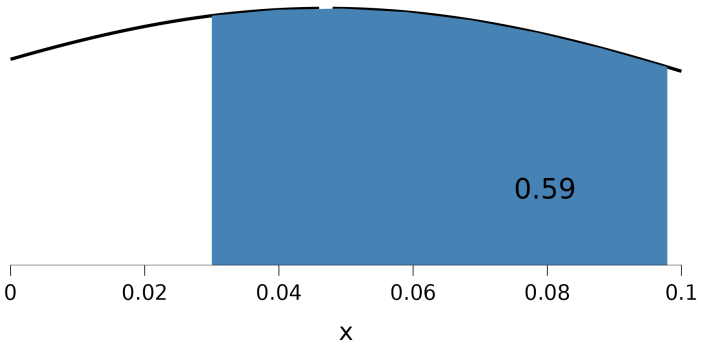

Figure 5.4: Stern nominal prob.=0.03 (n=128, k=86, l=2, p=1)

One the other hand, running another incident selecting this time $p = 2$

Code Listing 5.3: Stern Execution, p=2, l=2

```
python stern.py -c 40 -m 7 -t 6 -p 2   -l  2
```

we encounter nominal probability $Pr_{stern(p=2)} = 0.11$. For such a small difference of the p value there is a big difference on the corresponding probability having a good permutation. On the contrary, the complexity finding collisions between the L_1 and L_2 sets is greater when p is greater. Then, the corresponding variety of vectors in lists is increased, as the number of error bit increases, and so the number of calculations in Eq. 5.10 increases.

In a additional step, we run simultaneously the above two versions of Stern's configuration, let's denote that time as t_0. At a random time t_1, $t_1 >> t_0$, we had 40% success of revealing the error vector when setting $p = 1$ and 60% when setting $p = 2$. Thus, computation complexity does not affect negatively the total complexity $N_{Stern}(n,k,t,p,l)$, in small values of the block code length n. On the other hand, in such values, greater probability seems to determine more the results of the attack.

The overall switches explanation of the Stern's algorithm developed in python is listed in:

```
usage: stern.py [-h] [-c C] [-p P] [-l L] [-t T] [-m M]

options:
  -h, --help  show this help message and exit
  -c C        times to repeat the alg
  -p P        num of errors in k/2
  -l L        size of dimension l(stern)
  -t T        num of errors
  -m M        m in GF
```

5.4.2.3 Stern vs Prange

In probability domain, comparing the two algorithms , Plain-ISD (Prange) and Stern, for the same block code values ($n = 128$ and $k = 86$) we can clearly see a big difference between them. The $Pr_{pl-ISD} = 0.001$ whereas $Pr_{stern} = 0.11$. Running these two algorithms we indeed get the expected results. When Stern has revealed the error vector more than twenty times, Plain-ISD has not determined at all. The explanation is that when for high rates like $R = \frac{k}{n} = \frac{86}{128} \simeq 0.7$, the k-window is pretty large compared with the total block code length n. Thus, it is very difficult not to encounter any error bits in these coordinates, something that it is not allowed in Prange.

5.4.3 Ball-Collision-Decoding (BCD) Algorithm

In 2011, Bernstein, Lange and Peters [28] presented another information set decoding algorithm, they called it Ball-collision decoding (BCD). The general idea of BCD is very similar to the idea of Sterns algorithm but increases the success probability of one iteration by allowing an additional number of error bits within the fixed l coordinates. Therefore, BCD allows q additional 1s within the l-width window.

Ball-collision, like Stern's algorithm, splits the parity check matrix and the syndrome like

$$\tilde{H} = \begin{pmatrix} A & B & I_l & 0 \\ C & D & 0 & I_{n-k-l} \end{pmatrix}$$
$$\tilde{s} = \begin{pmatrix} \tilde{s}_l & \tilde{s}_R \end{pmatrix}$$

where $l = l_1 + l_2$. But, at the l_1, l_2 coordinates one may encounter q_1, q_2 errors, respectively, where $0 \leq q_1 + q_2 \leq t - p_1 + p_2$.

Bernstein uses the term collision-decoding as a the special occasion of ball-collision-decoding when $q_1 = q_2 = 0$ and $p_1 = p_2$, likewise Stern. If q_1, $q_2 \neq 0$, then the Eq. 5.9 must include the middle part of then non zero vectors. Thus, for all these vectors $\tilde{e_{L_1}}$, $\tilde{e}_{L_2}$, $\tilde{e_{l_1}}$, $\tilde{e}_{l_2}$ with $wt(\tilde{e}_{L_1}) = p1$, $wt(\tilde{e}_{L_2}) = p2$, $wt(\tilde{e}_{l_1}) = q_1$, $wt(\tilde{e}_{L_1}) = q_2$, since is fulfilled the left part of the syndrome will be

$$\begin{aligned} \tilde{s_l} &= \tilde{e}_L U^T + \tilde{e}_l I_l \\ &= \tilde{e}_{L_1} A^T + \tilde{e}_{L_2} B^T + \tilde{e}_{l_1} I_{l_1} + \tilde{e}_{l_2} I_{l_2} \\ &= \tilde{e}_{L_1} A^T + \tilde{e}_{L_2} B^T + \tilde{e}_{l_1} + \tilde{e}_{l_2} \end{aligned}$$

or finally

$$\tilde{s_l} = \tilde{e}_{L_1} A^T + \tilde{e}_{L_2} B^T + \tilde{e}_{l_1} + \tilde{e}_{l_2} \tag{5.11}$$

Then, we can compute syndrome's right part $\tilde{s}_R$ as usual, following the same rational as Stern does (see Alg. A.1.5).

5.4.3.1 Complexity

At the perspective of success probability let us assume that e is a random vector of weight t. One iteration of the ball-collision decoding may find the e if it has the right weight distribution. So, the weight should be distributed like:

(i) p_1 in the first k_1 positions specified by the information set,

(ii) p_2 in the remaining k_2 positions specified by the information set,

(iii) q_1 on the first l_1 positions outside the information set,

(iv) q_2 on the next l_2 positions outside the information set.

The probability that e has this weight distribution is exactly

$$Pr_{BCD}(n,k,t,p_1,p_2,q_1,q_2,l_1,l_2) = \binom{n-k-l_1-l_2}{t-p_1-p_2-q_1-q_2}\binom{k_1}{p_1}\binom{k_2}{p_2}\binom{l_1}{q_1}\binom{l_2}{q_2}\binom{n}{t}^{-1}$$

In this occasion, the computational cost for each iteration is again both the Gauss elimination at the $n-k$ coordinates $T_{GE}(n-k)$ and the manipulation of each list containing errors. Thus, $C_{BCD} = T_{GE}(n-k) + |L_1^{k/2}| + |L_2^{k/2}| + |L_3^{k/2}| + |L_4^{k/2}|$. We can denote that

$$N_{BCD}(n,k,t,p_1,p_2,q_1,q_2,l_1,l_2) = \frac{C_{BCD}}{Pr_{BCD}} \tag{5.12}$$

5.4.3.2 BCD vs Stern

From the perspective of ball-collision decoding, the fundamental disadvantage of Stern's decoding algorithm is that errors are required to avoid an asymptotically quite large stretch of $l_1 + l_2$ positions. Ball-collision decoding makes a much more reasonable hypothesis, namely that there are asymptotically increasingly many errors in those positions. It requires extra work to enumerate these positions but the extra work is only about the square root of the improvement in success probability.

Moreover, BCD has faster inner loop. For instance, if Stern algorithm tries to compute the left part of the syndrome $\tilde{s}$ (see Eq. 5.9), then the terms $\tilde{e}_{L_1}A^T$, $\tilde{e}_{L_2}A^T$ must be computed. In this case, it is more expensive to iterate those list having the same amount of bit errors. For example, if $p = 2$ and $\frac{k}{2} = k_1 = 40$ then we would have $\binom{40}{2}$ =780 different vectors in list L_1, and the same 780 vectors in list L_2, to multiply with the matrix A^T, simultaneously. Instead, using BCD configuration with $p_1 = 1$ and $p_2 = 2$, we could have only 40 different vectors in L_1, which means 740 less multiplications with A^T.

Assume now that the remaining error bit $p - p_1 = 1$ goes to the l_1 part. Let also say that l_1=8. Then, we will have only $\binom{8}{1}$ =8 different vectors $\tilde{e}_{l_1}$, prompting the term $\tilde{e}_{l_1}$ to the Eq. 5.11. Consider that the addition operation has constant time complexity. So, Bernstein replace a complex process, the multiplication, to fewer, less complex.

For experimental purpose, let us consider the same Goppa characteristics for a McEliece cryptosystem we also used in Stern vs Prange comparison (5.4.2.3), hence $n = 128$ and $k = 86$. Then, we select to attack with the optimal configuration of Stern alg. $(p, l) = (2, 2)$. For BCD we use the configuration $(p_1, p_2) = (2, 1)$, $q_1 = q_2 = 0$ and $l_1 = l_2 = 1$.

Code Listing 5.4: Stern

```
python stern.py -c 40 -m 7 -t 6 -p 2   -l  2
```

Code Listing 5.5: BCD

```
python ballColl.py -c 40 -m 7 -t 6 -p1 2 -p2 1  -l1 1 -l2 1 -q1 -1 -
    q2 -1
```

NOTE: In order to parse zero values we declare them as a negative value, for example -1.

For further details considering the usage of the ball-collision algorithm:

```
usage: ballColl.py [-h] [-c C] [-l L] [-p1 P1] [-q1 Q1] [-t T] [-m M]

optional arguments:
  -h, --help  show this help message and exit
  -c C        Times to repeat the alg
  -l L        size of dimension l(stern)
  -p1 P1      num of errors in k1
  -q1 Q1      num of errors in l1
  -t T        num of errors
  -m M        m in GF
```

After forty repetitions of each Stern and BCD, we jotted the following values of the Tab. 5.3, considering the probabilities.

Table 5.3: BCD vs Stern success probability

Alg.	Nom. prob.	Exp. prob.
Stern	0.11	0.077
BCD	0.07	0.068

As BCD success nominal probability is less than Stern's, we must strongly consider the complexity of the operations that Stern has to do in order to reveal the error vector. Embracing the aforementioned reasoning in Ch. 5.4.3.1 we can easily deduct that there is an enormous difference in complexity when both algorithms try to find the collisions between the lists L_1 and L_2.

In Fig. 5.5 and 5.6 are represented the experimental probabilities when run the algorithms with the parameters aforementioned. BCD experimental probability is equivalent to nominal, with the probability to get a good permutation greater that the mean value. On the other hand, Stern is more unlikely to reach a good permutation. Its experimental probability differs from the nominal. If we add th high complexity, we may deduct that Stern is significantly slower than BCD.

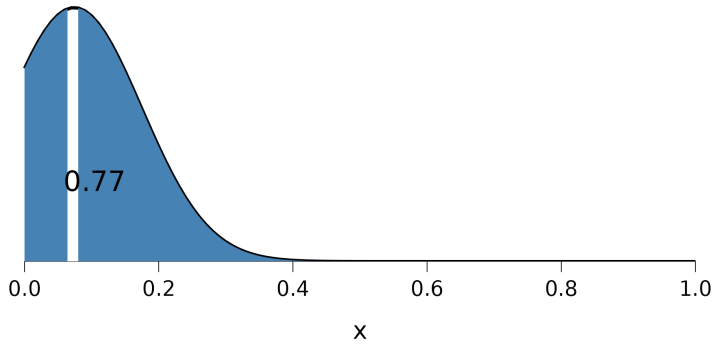

Figure 5.5: Stern exper. prob.=0.077

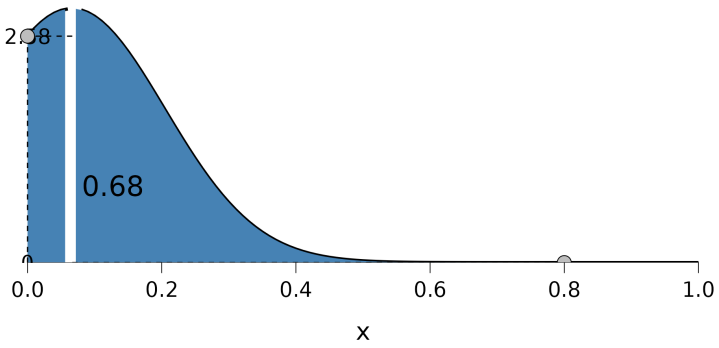

Figure 5.6: BCD exper. prob.=0.068

We can verify the above conclusions investigating the time execution. In order to do so, we run multiple times the algorithms, measuring the time of their run time. Table 5.4 represents an independent sample t-test of BCD vs Stern. Due to high time execution, we could not run Stern as multiple times as BCD. Though, there is a capable sample which gives a solid information about time complexity. So, we can clearly see that they have a significant mean time execution difference. BCD mean time value until it gets a good permutation, is remarkably less than Stern's. Additionally, Stern has higher coefficient of variation. Thus, the dispersion is large. This implies the uncertainty of having a fix time in order to get a good permutation. Fig. 5.7 monitors the compression and dispersion of running time for both algorithms.

Table 5.4: Time execution comparison

	Group	N	Mean	SD	SE	Coefficient of variation
Time	BCD	43	4029.698	2740.040	417.852	0.680
	Stern	43	15602.116	13607.779	2075.167	0.872

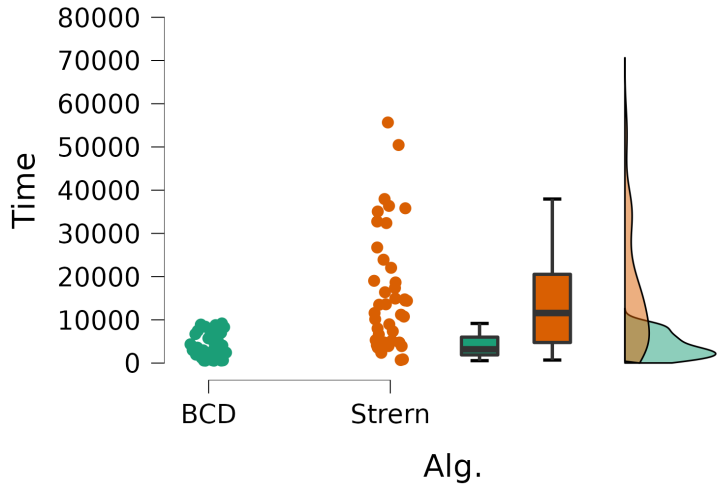

Figure 5.7: Time execution comparison

5.5 Summary

While RSA cryptosystem is the major representative in the public key field, its core might be in a really hazard. The hardness of prime numbers factorisation might be extinguished with the arrival of the quantum computers. The McEliece cryptosystem, which is based in coding theory, is an valuable alternative.

In chapter 5 there has been an effort to order the attacks against the McEliece cryptosystem in the manner of effectiveness. The wide, well known brute-force attack method seems weak against large McEliece keys. Unfortunately, its resistance is

also its weakness. Keys with large size set their usage forbidden. But, regarding the quantum attacks, using that large keys may be key factor for safety in the future.

On the other hand, several coding theory algorithms appear capable to attack the McEliece cryptosystem. These algorithms belongs at the Information-Set-Decoding family. The most remarkable of these attacks is the effectiveness compared to the Grover's algorithm, which is the latest known in quantum computing.

In this chapter we analyzed three of Information-Set-Decoding candidates, begging from the very precestor, Prange's algorithm, in Ch. 5.4.1. Running the algorithm for several rates $R = \frac{k}{n}$, between $(0, 1)$, we clearly verified that the method is more effective :

(i) in low rates, where the message length is small compared to the codeword's length whilst the number of errors in the error vector are large

(ii) in high rates, where the message length is large and the errors are few.

Nevertheless, for mid rates, let's say $R = \frac{k}{n} = 0.4$, we measured experimental probability of finding a good permutation P similar to its nominal value (see Ch. 5.4.1.1), in order to reveal the error vector.

The next experiment concerns Stern's algorithm, which is an improvement of Prange's. Stern distributes more the errors along the investigated error vector. This consideration increases the possibility to bump to the desired permutation matrix. In Ch. 5.4.2.3 we saw that Stern has revealed the error vector more than twenty times when Plain-ISD (prange) has not determined at all. Though we are able to have better results using Stern, we firstly must configure it suitable. In other words we should guess a proper error distribution. Bad Stern configurations may lead us to worst results comparing Prange's alg.

The next attempt was to investigate an algorithm developed several years later. Berstein et al, expand more the error distribution. Their paper permits error existence in the previous l-length zero-window. Likewise Stern, Berstein algorithm also named Ball-Collision-Decoding (BCD), has its own best case configurations. In some of them, where the errors are few, BCD may operate with the l-length zero-window on. If the key size is large, more errors occur and then is more likely to appear few errors in the l-length part. Nevertheless, even in smaller keys, BCD promises better results comparing to Stern. Though in Tab. 5.3 we notice that BCD has success nominal probability less than Stern's, we must strongly consider the complexity of the operations that Stern has to do in order to reveal the error vector. This becomes true when we calculate execution time of each algorithm. In Fig. 5.7 we are able to notice that difference.

CHAPTER 6

The Future of Post Quantum Security (PQS)

6.1 NIST Contest for PQS Standardization

NIST had initiated a process to develop and standardize one or more additional public-key cryptographic algorithms. As a first step in this process, NIST requested public comment on draft minimum acceptability requirements, submission requirements, and evaluation criteria for candidate algorithms. Comments received are posted at https://www.nist.gov/pqcrypto, along with a summary of the changes made as a result of these comments. The purpose of this notice is to announce that nominations for post-quantum candidate algorithms may now be submitted, up until the final deadline of November 30, 2017. The main cause of the submissions was NIST to choose some algorithms for standardization in both signatures and public key cryptography.

During the third round, the finalists were the first seven, and the term 'alternate' was introduced for the other eight algorithms. The finalists would continue to be reviewed for consideration for standardization at the conclusion of the third round. McEliece cryptosystem was one of the finalist. NIST had expected to have a fourth round of evaluation for some of the candidates on this track. Several of these alternate candidates had worse performance than the finalists but might be selected for standardization based on a high confidence in their security. Other candidates had acceptable performance but require additional analysis or other work to inspire sufficient confidence in their security or security rationale. In addition, some alternates were selected based on NISTs desire for a broader range of hardness assumptions in future post-quantum security standards, their suitability for targeted use cases, or their potential for further improvement.

NIST has completed the third round of the Post-Quantum Cryptography (PQC) standardization process which selects public-key cryptographic algorithms to protect information through the advent of quantum computers. A total of four candidate algorithms have been selected for standardization, and four additional algorithms will continue into the fourth round. So, after careful consideration during the third round of the NIST PQC Standardization Process, NIST has identified four candidate algorithms for standardization and will recommend two primary algorithms to

be implemented for most use cases: CRYSTALS-KYBER (key-establishment) and CRYSTALS-Dilithium (digital signatures). In addition, the signature schemes FALCON and SPHINCS+ will also be standardized.

Classic McEliece was indeed a finalist but is not being standardized at this time. Although Classic McEliece is widely regarded as secure, NIST does not anticipate it being widely used due to its large public key size. NIST may choose to standardize Classic McEliece at the end of the fourth round. Nevertheless, McEliece is strong candidate for PQC. In Ch. 4.5 are listed several applications of this system that have been studied even in hardware implementation in order to avoid the drawback of the key size.

6.2 Research: Horizon 2020

The European Commission has also promoted the research over post-quantum cryptosystems. A European research group, PQCRYPTO, has been funded by the European Union Horizon 2020 project and is conducting research on post-quantum cryptography for small devices, the Internet and the cloud. Another project supported by Horizon 2020 is SAFEcrypto which focuses on practical and physically secure post-quantum cryptographic solutions in protecting satellite and public safety communication systems, as well as preserving the privacy of data collected by the government.

6.3 The Rise of the IoT

The Internet of Things (IoT)-centric concepts like augmented reality, high-resolution video streaming, self-driven cars, smart environment, e-health care, etc. have a ubiquitous presence now. These applications require higher data rates, large bandwidth, increased capacity, low latency and high throughput. In light of these emerging concepts, IoT has revolutionized the world by providing seamless connectivity between heterogeneous networks (HetNets).

The eventual aim of IoT is to introduce the plug and play technology providing the end user, ease of operation, remotely access control and configurability. Fifth Generation (5G) cellular networks provide key enabling technologies for ubiquitous deployment of the IoT technology [29].

The enterprise IoT market grew 22.4% to $157.9 billion in 2021, according to the March 2022 update of IoT Analytics Global IoT Enterprise Spending Dashboard (see Fig. 6.1). The market grew slightly slower than the 24% forecasted last year due to several factors, including a slower-than-anticipated overall economic recovery, a lack of chipsets and disrupted supply chains. North America was the fastest growing region in 2021 (+24.1%), and process manufacturing was the fastest-growing segment (+25%).

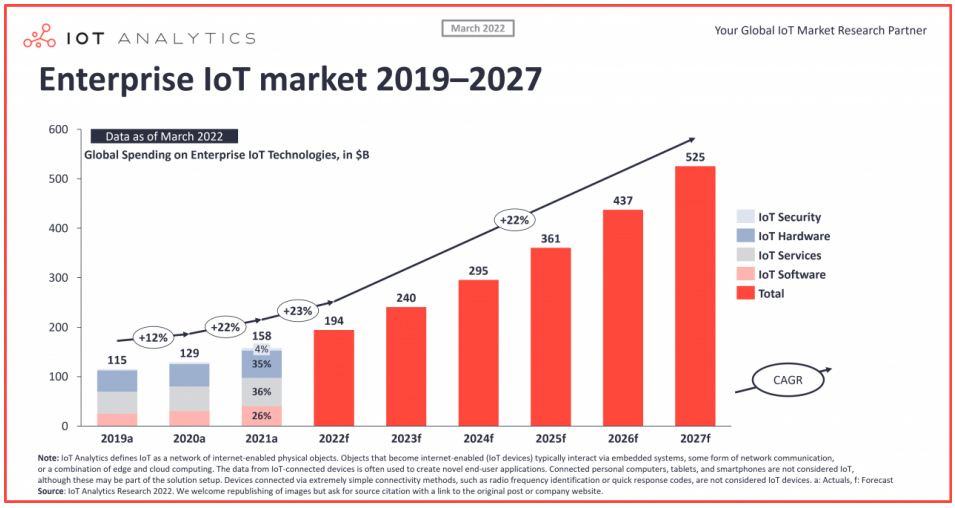

Figure 6.1: IoT growth

At this point, IoT Analytics forecasts the IoT market size to grow at a CAGR of 22.0% to $525 billion from 2022 until 2027. The five-year forecast has been lowered from the previous year. A number of growth headwinds have had a much more profound impact than previously anticipated, namely supply shortages and disruptions (most notably chip shortages which are now expected to extend well into 2024 and possibly even beyond) and labor shortages, especially for sought-after software jobs. Despite the lowered growth projections, IoT remains a very hot technology topic with many projects moving into the rollout phase. The number of connected IoT devices is expected to reach 14.5 billion globally by the end of 2022 [31].

6.4 IoT PQS

Generally, the main security goals for the IoT are confidentiality, integrity, and authentication. Confidentiality guarantees that sensitive information cannot be leaked to unauthorized entities, while integrity prevents information from being modified en route. Finally authentication ensures that the communicating entities are indeed those they declare to be [30].

To achieve the aforementioned security goals regarding the IoT, these protocols use cryptographic primitives such as the Advanced Encryption Standard (AES) for confidentiality and integrity and elliptic curve cryptosystems (ECCs), which include the Elliptic Curve Digital Signature Algorithm (ECDSA) for integrity and authentication and also the Elliptic Curve Diffie-Hellman (ECDH) algorithm for exchanging keys used in AES. However, recent advances in quantum computing threaten the security of the current IoT using these cryptographic schemes as long as the security of Rivest-Shamir-Adleman (RSA) and Diffie-Hellman (DH) key exchange schemes are based on the difficulty of solving some number-theoretic problems such as integer

factorization and discrete logarithms.

The security of the ECC is based on the difficulty of solving the elliptic curve discrete logarithm problem. As early as 1994, mathematician Peter Shor of Bell Laboratories showed that quantum computers can solve the integer factorization problem, as aforementioned in Ch. 2.4 and the discrete logarithm problems (elliptic curve) in an efficient way, sparking great research interest in quantum computing area. Since then, quantum algorithms like Grover's search algorithm have been proposed (see Ch. 5.2), which provide significant speedup for many problems. Other examples include the quantum algorithms using the quantum Fourier transform, the quantum walk for solving searching problems and adiabatic quantum computing for optimization problems. Besides that, much research is performed on how to design and build more powerful quantum computers with less resources to implement these algorithms.

Even though there are quantum secure replacements for the cryptographic standards in use today, it will take a long time for the transition from currently used IoT systems to their quantum-resistant counterparts. Regarding the fact that we are at the very beginning of the standardization process for quantum resistant algorithms and research on their application in the IoT is limited, it is urgent to make significant efforts in securing IoT systems against possible attacks by quantum computers. Therefore, no matter whether we can predict the exact arrival time of large-scale quantum computers, we should act now to prepare IoT systems for the quantum world.

CHAPTER 7

Conclusions

Based on the premature work upon the quantum computers, the concept of Post Quantum Security is optimistic. The duration of analysis of the McEliece cryptosystem may indicated that it is a good candidate as a panakeia of the Quantum attacks. Still, the RSA system is a very good method, alongside with other methods such as eliptic curves. It is pretty sad that McEliece cryptosystem is not included of the NIST filal chose, at the time being. Nevertheless, it has a lot of potential to thrive in future. If efforts to reduce the size of the public key come in hand, then in next contest it is sure that will be one of the finals.

Information Set Decoding algorithms has their roots back in 1962. Now, in 2022, we discuss proposals that evolve the initial report. We saw that the ball-collission algorithm is steadily a pioneer in decoding the linear codes in which McEliece cryptosystem is based. Furthermore, BCD alg. hints that as long as the key increases it seems that these algorithms become more sophisticated and sieve the increase spread of the errors. On the other hand, if the McEliece cryptosystem manages to work as a hardware implementation, it is obvious that such decoding algotithms are not effective.

APPENDIX A

Appendix

A.1 Algorithms in Pseudo-code

A.1.1 McEliece Encryption

Algorithm A.1 McEliece Encryption

input: a message m over the binary GF with length k, the public key $G' = SGP$ and the parameter t

output: the ciphertext c, over that GF with length n

1: compute mG'

2: choose a random error vector e, over GF with length n and weight $wt(e) = t$

3: compute

$$c = mG' + e \tag{A.1}$$

4: return c

A.1.2 McEliece Decryption

Algorithm A.2 McEliece Decryption

input: a ciphertext c, over GF with length n and the secret key (S, G, P)

output: the message m over the binary GF, with length k

1: compute P^{-1}, which is also a permutation matrix

2: compute $c' = cP^{-1}$, which is another codeword with the same weight

3: compute $e' = eP^{-1}$, which is another error vector with the same weight

4: substitute functions in steps 2,3 to A.1, which becomes

$$c' = mSG + e' \tag{A.2}$$

5: the mS is another $1 \times k$ message m' when multiplied with the generator matrix G. The first k coordinates of the mSG is the $m' = mS$.

6: compute $m = m'S^{-1}$

A.1.3 Plain-ISD (Prange)

Algorithm A.3 Plain-ISD

input: Public version of the parity check matrix H (H_{pub}), the codeword c and the weight t

output: Error vector e with $eH^T = s$ and $wt(e) = t$

compute $s = cH^T$
found=false
while not found **do**
    select permutation matrix P, randomly
    apply Gauss Elimination to HP until get $\tilde{H} = (U\, I_{n-k})$, where $\tilde{H} = QHP$
    set $\tilde{s} = sQ^T$
    if $wt(\tilde{s}) = t$ **then**
        found=true
        set $\tilde{e} = (0\, \tilde{s})$
    end
end
set $e = \tilde{e}P^T$
return e

A.1.4 Stern

Algorithm A.4 Stern's Algorithm

input: Public version of the parity check matrix H (H_{pub}), the codeword c, the weight t and the parameters p, l

output: Error vector e with $eH^T = s$ and $wt(e) = t$

compute $s = cH^T = \begin{pmatrix} s_l & s_R \end{pmatrix}$
found1=found2=false
while not found1 **do**
    select permutation matrix P, randomly
    apply Gauss Elimination to HP until get $\tilde{H} = \begin{pmatrix} U & I_{n-k} \end{pmatrix} = QHP$
    Split parity check matrix as

$$\tilde{H} = \begin{pmatrix} A & B & I_l & 0 \\ C & D & 0 & I_{n-k-l} \end{pmatrix}$$

    create list L_1 with all vectors with weight p and length $\frac{k}{2}$
    create list L_2 with all vectors with weight p and length $\frac{k}{2}$
    for all $\tilde{e}_{L_1}, \tilde{e}_{L_2} \in L_1, L_2$ **do**
        if $\tilde{s}_l = \tilde{e}_{L_1} A^T + \tilde{e}_{L_2} B^T$ **then**
            set found2=true
            set $\tilde{e}_R = \tilde{s}_R + \tilde{e}_{L_1} C^T + \tilde{e}_{L_2} D^T$
            set $\tilde{e} = \begin{pmatrix} \tilde{e}_{L_1} & \tilde{e}_{L_2} & 0 & \tilde{e}_R \end{pmatrix}$
        end
        break if found2=true
    end
    if $wt(\tilde{e}) = t$ and found2=true **then**
        set found1=true
    end
    break if found1=true
end
set $e = \tilde{e}P^T$
return e

A.1.5 Ball-Collision Decoding (BCD)

Algorithm A.5 Ball-Collision Algorithm

input: Public version of the parity check matrix H (H_{pub}), the codeword c, the weight t and the parameters $p_1, p_2, q_1, q_2, k_1, k_2, l_1, l_2 \in \mathbb{Z} : 0 \leq k_1, 0 \leq k_2, k = k_1 + k_2, 0 \leq p_1 \leq k_1, 0 \leq p_2 \leq k_2, 0 \leq q_1 \leq l_1, 0 \leq q_2 \leq l_2, 0 \leq t - p_1 - p_2 - q_1 - q_2 \leq n - k - l_1 - l_2$.

output: Error vector e with $eH^T = s$ and $wt(e) = t$

1: compute $s = cH^T = \begin{pmatrix} s_l & s_R \end{pmatrix}$
2: found1=found2=false
3: **while** not found1 **do**
4: select permutation matrix P, randomly
5: apply Gauss Elimination to HP until get $\tilde{H} = \begin{pmatrix} U & I_{n-k} \end{pmatrix} = QHP$
6: Split parity check matrix as

$$\tilde{H} = \begin{pmatrix} A & B & I_l & 0 \\ C & B & 0 & I_{n-k-l} \end{pmatrix}$$

7: create list L_1 with all vectors with weight p_1 and length k_1
8: create list L_2 with all vectors with weight p_2 and length k_2
9: create list l_1 with all vectors with weight q_1 and length l_1
10: create list l_2 with all vectors with weight q_2 and length l_2
11: **for** all $\tilde{e}_{L_1}, \tilde{e}_{L_2}, \tilde{e}_{l_1}, \tilde{e}_{l_2} \in L_1, L_2, l_1, l_2$ **do**
12: **if** $\tilde{s}_l = \tilde{e}_{L_1} A^T + \tilde{e}_{L_2} B^T + \tilde{e}_{l_1} + \tilde{e}_{l_2}$ **then**
13: set found2=true
14: set $\tilde{e}_R = \tilde{s}_R + \tilde{e}_{L_1} C^T + \tilde{e}_{L_2} D^T$
15: set $\tilde{e} = \begin{pmatrix} \tilde{e}_{L_1} & \tilde{e}_{L_2} & \tilde{e}_{l_1} & \tilde{e}_{l_2} & \tilde{e}_R \end{pmatrix}$
16: **end**
17: break if found2=true
18: **end**
19: **if** $wt(\tilde{e}) = t$ and found2=true **then**
20: set found1=true
21: **end**
22: break if found1=true
23: **end**
24: set $e = \tilde{e}P^T$
25: return e

A.2 McEliece Cryptosystem in Python

A.2.1 Irreducible polynomial

```
def get_irreducible(t_val):
    x = sympy.Symbol('x')
    # Randomly generate a polynomial of degree t
    # If it is reducible, generate another
    while True:
        # Coefficient of x^t must be 1
        polylist = [1]
        # Randomly select coefficients
        for i in range(t_val-1):
            polylist.append(random.randint(0, 1))
        # Coeffiecient of x^0 must be 1
        polylist.append(1)
        if (sum(polylist) % 2) == 1:
            # Produce a polynomial from the list
            p = sympy.Poly(polylist, x, domain='FF(2)')
            # Check that this is irreducible
            if p.is_irreducible:
                # Return the coefficients in a list
                return(p.all_coeffs())
```

A.2.2 Parity Check Matrix H

```
def get_H(m_val, t_val, k_vec, g_vec):
    support = []
    g_a = []
    k_poly = sympy.Poly(k_vec, alpha, domain='FF(2)')
    # Store the support of the code into a list
    for i in range(pow(2, m_val)):
        if i == 0:
            # Calculate g(0)
            a_poly = sympy.reduced(sympy.Poly(sympy.Poly(g_vec, x,
            domain='FF(2)').eval(0), alpha, domain='FF(2)').args[0],
            [k_poly])[1].set_modulus(2)
        else:
            # Calculate g(a^(i-1))
            a_poly = sympy.reduced(sympy.Poly(g_vec, alpha**(i-1),
            domain='FF(2)').args[0], [k_poly])[1].set_modulus(2)
        if not a_poly.is_zero:
            # Only store if it is not zero
            if i == 0:
                support.append(sympy.reduced(0, [k_poly])[1].set_modulus(2))
            else:
                support.append(sympy.reduced(alpha**(i-1),
                        [k_poly])[1].set_modulus(2))
            g_a.append(a_poly)
    if args.vv: print('\nSupport:\n', support, '\n\ng(a_n):\n', g_a)
    col = []
```

```
    # Store the inverses of g(a)
    for element in g_a:
        inverse = sympy.invert(element, k_poly)
        col.append(inverse)
    if args.vv: print('\nInverses:\n', col)
    # Form the Parity check matrix
    poly_H = []
    for i in range(t_val):
        poly_H_row = []
        for j in range(len(support)):
            top = sympy.Poly.pow(support[j], i)
            product =  sympy.reduced(sympy.Poly.mul(top, col[j]),
                        [k_poly])[1].set_modulus(2)
            poly_H_row.append(product)
        poly_H.append(poly_H_row)
    bin_H = sympy.zeros(t_val * m_val, len(support))
    # Turn the parity check matrix into a binary matrix
    for i in range(t_val):
        for j in range(len(support)):
            current_poly = poly_H[i][j].all_coeffs()
            current_len = len(current_poly)
            for k in range(current_len):
                try:
                    bin_H[(i*(m_val))+k,j] = current_poly[current_len-k-1]
                except:
                    sympy.pprint(bin_H)
                    print('i =', i, ', j =', j, ', k =', k)
                    exit()
    bin_H, pivot = bin_H.rref(iszerofunc=lambda x: x % 2==0)
    bin_H = bin_H.applyfunc(lambda x: mod(x,2))
    bin_H = fixup_H(bin_H, pivot)
    return(bin_H)
```

A.2.3 Generator Matrix G

```
# From the parity check matrix, create the Generator matrix
def get_G(bin_H):
    for i in range(bin_H.shape[0]):
        bin_H.col_del(0)
    bin_G = bin_H.T
    ident = sympy.eye(bin_G.shape[0])
    for i in range(bin_G.shape[0]):
        bin_G = bin_G.col_insert(bin_G.shape[1], ident.col(i))
    return(bin_G)
```

A.2.4 Key Generation

```
#!/usr/bin/python3
import random
```

```
3  import sympy
4  import numpy
5  from McElieceUtil import *
6  #from InfoSetUtilities import *
7
8
9  def myReadFromFile(filename):
10     with gzip.open(filename, 'rb') as f:
11         matrix= sympy.Matrix(pickle.loads(f.read()))
12     return matrix
13 x = sympy.Symbol('x')
14 alpha = sympy.Symbol('a')
15 # Generate key pairs
16 def keygen(m_val, t_val, files):
17     global x
18
19     # Random irreducible polynomial of degree m
20     k_vec = get_irreducible(m_val)
21
22     if args.v: print("k(x) = ", sympy.Poly(k_vec, x, domain='FF(2)'))
23     # Random irreducible polynomial of degree t
24     g_vec = get_irreducible(t_val)
25     if args.v: print("g(x) = ", sympy.Poly(g_vec, x, domain='FF(2)'))
26     # Produce k*n generator matrix G for the code
27     H_matrix = get_H(m_val, t_val, k_vec, g_vec)
28     if args.v: print('\nH ='); sympy.pprint(H_matrix); print(H_matrix.shape)
29     G_matrix = get_G(H_matrix[:,:])
30     if args.v: print('\nG ='); sympy.pprint(G_matrix); print(G_matrix.shape)
31     k_val = G_matrix.shape[0]
32     # Select a random k*k binary non-singular matrix S
33     S_matrix = get_nonsingular(k_val)
34     if args.v: print('\nS ='); sympy.pprint(S_matrix)
35     # Select a random n*n permutation matrix P
36     n_val = G_matrix.shape[1]
37     permutation = random.sample(range(n_val), n_val)
38     P_matrix = sympy.Matrix(n_val, n_val,
39                 lambda i, j: int((permutation[i]-j)==0))
40     if args.v: print('\nP ='); sympy.pprint(P_matrix)
41     # Compute k*n matrix G_pub = SGP
42     G_pub = (S_matrix * G_matrix * P_matrix).applyfunc(lambda x: mod(x,2))
43     H_pub=(H_matrix*P_matrix).applyfunc(lambda x: mod(x,2))
44     '''
45     #my code to turn Gpub to systematic
46     isSystem, G_pub_Sys=Gauss_Elim(G_pub, G_pub.shape[1]-G_pub.shape[0],G_pub.shape[
                                                    1])
47     if isSystem:
48         print('G_pub turned systematic.')
49         G_pub=G_pub_Sys
50     else:
51         print('Unable to systematic G_pub, exit..')
52         exit()
53     '''
```

```
54      if args.v: print('\nG_pub ='); sympy.pprint(G_pub); print(G_pub.shape)
55      # Public key is (G_pub, t)
56      # Private key is (S, G, P)
57      # length of the Goppa Code
58      if args.v: print("\nn = ", n_val)
59      if args.v: print("k =", k_val)
60      writeKeys(G_matrix,G_pub, t_val, S_matrix, H_matrix,H_pub, P_matrix, files)
61
62  # From the parity check matrix, create the Generator matrix
63  def get_G(bin_H):
64      for i in range(bin_H.shape[0]):
65          bin_H.col_del(0)
66      bin_G = bin_H.T
67      ident = sympy.eye(bin_G.shape[0])
68      for i in range(bin_G.shape[0]):
69          bin_G = bin_G.col_insert(bin_G.shape[1], ident.col(i))
70      return(bin_G)
71  def fixup_H(bin_H, pivot):
72      num_removed = 0
73      for j in range(bin_H.shape[0]):
74          if bin_H.row(j-num_removed) == sympy.zeros(1, bin_H.shape[1]):
75              bin_H.row_del(j-num_removed)
76              num_removed += 1
77      for i in range(bin_H.shape[0]):
78          if not i == pivot[i]:
79              col_a = bin_H.col(i)
80              col_b = bin_H.col(pivot[i])
81              bin_H.col_del(i)
82              bin_H = bin_H.col_insert(i, col_b)
83              bin_H.col_del(pivot[i])
84              bin_H = bin_H.col_insert(pivot[i], col_a)
85      return(bin_H)
86  # Create a parity check matrix give then Goppa code information
87  def get_H(m_val, t_val, k_vec, g_vec):
88      support = []
89      g_a = []
90      k_poly = sympy.Poly(k_vec, alpha, domain='FF(2)')
91      # Store the support of the code into a list
92      for i in range(pow(2, m_val)):
93          if i == 0:
94              # Calculate g(0)
95              a_poly = sympy.reduced(sympy.Poly(sympy.Poly(g_vec, x,
96              domain='FF(2)').eval(0), alpha, domain='FF(2)').args[0],
97              [k_poly])[1].set_modulus(2)
98          else:
99              # Calculate g(a^(i-1))
100             a_poly = sympy.reduced(sympy.Poly(g_vec, alpha**(i-1),
101             domain='FF(2)').args[0], [k_poly])[1].set_modulus(2)
102         if not a_poly.is_zero:
103             # Only store if it is not zero
104             if i == 0:
105                 support.append(sympy.reduced(0, [k_poly])[1].set_modulus(2))
```

```
106             else:
107                 support.append(sympy.reduced(alpha**(i-1),
108                         [k_poly])[1].set_modulus(2))
109             g_a.append(a_poly)
110     if args.vv: print('\nSupport:\n', support, '\n\ng(a_n):\n', g_a)
111     col = []
112     # Store the inverses of g(a)
113     for element in g_a:
114         inverse = sympy.invert(element, k_poly)
115         col.append(inverse)
116     if args.vv: print('\nInverses:\n', col)
117     # Form the Parity check matrix
118     poly_H = []
119     for i in range(t_val):
120         poly_H_row = []
121         for j in range(len(support)):
122             top = sympy.Poly.pow(support[j], i)
123             product =  sympy.reduced(sympy.Poly.mul(top, col[j]),
124                         [k_poly])[1].set_modulus(2)
125             poly_H_row.append(product)
126         poly_H.append(poly_H_row)
127     bin_H = sympy.zeros(t_val * m_val, len(support))
128     # Turn the parity check matrix into a binary matrix
129     for i in range(t_val):
130         for j in range(len(support)):
131             current_poly = poly_H[i][j].all_coeffs()
132             current_len = len(current_poly)
133             for k in range(current_len):
134                 try:
135                     bin_H[(i*(m_val))+k,j] = current_poly[current_len-k-1]
136                 except:
137                     sympy.pprint(bin_H)
138                     print('i =', i, ', j =', j, ', k =', k)
139                     exit()
140     bin_H, pivot = bin_H.rref(iszerofunc=lambda x: x % 2==0)
141     bin_H = bin_H.applyfunc(lambda x: mod(x,2))
142     bin_H = fixup_H(bin_H, pivot)
143     return(bin_H)
144 def get_nonsingular(k_val):
145     # Randomly generate a k*k Matrix
146     # If it is singular generate another
147     while True:
148         matrix = sympy.Matrix(numpy.random.choice([sympy.Integer(0),
149         sympy.Integer(1)], size=(k_val, k_val), p=[1./3, 2./3]))
150         if matrix.det() % 2:
151             return matrix
152 def get_irreducible(t_val):
153     x = sympy.Symbol('x')
154     # Randomly generate a polynomial of degree t
155     # If it is reducible, generate another
156     while True:
157         # Coefficient of x^t must be 1
```

```
158         polylist = [1]
159         # Randomly select coefficients
160         for i in range(t_val-1):
161             polylist.append(random.randint(0, 1))
162         # Coeffiecient of x^0 must be 1
163         polylist.append(1)
164         if (sum(polylist) % 2) == 1:
165             # Produce a polynomial from the list
166             p = sympy.Poly(polylist, x, domain='FF(2)')
167             # Check that this is irreducible
168             if p.is_irreducible:
169                 # Return the coefficients in a list
170                 return(p.all_coeffs())
171 # From https://stackoverflow.com/questions/31190182/sympy-solving-matrices
172 # -in-a-finite-field
173 def mod(x, modulus):
174     numer, denom = x.as_numer_denom()
175     try:
176         return numer*sympy.mod_inverse(denom,modulus) % modulus
177     except:
178         print('Error: Unable to apply modulus to matrix')
179         exit()
```

A.2.5 Write Keys in Files

```
1  def writeKeys(Gen_matrix,G_matrix, t_val, S_matrix, H_matrix,H_pub, P_matrix,
                                          filename):
2      with gzip.open(filename + '.pub', 'wb') as f:
3          f.write(pickle.dumps([G_matrix, t_val]))
4      with gzip.open(filename + '.priv', 'wb') as f:
5          f.write(pickle.dumps([S_matrix, H_matrix, P_matrix, t_val]))
6      with gzip.open(filename + '.parity', 'wb') as f:
7          f.write(pickle.dumps([H_matrix]))
8      with gzip.open(filename + '.gen', 'wb') as f:
9          f.write(pickle.dumps([Gen_matrix]))
10     with gzip.open(filename + '.Hpub', 'wb') as f:
11         f.write(pickle.dumps([H_pub]))
```

A.2.6 Encryption

```
1  def encrypt(mfile, pubkey,cipherFile):
2      G_pub, t_val = readFromFile(pubkey)
3      k=G_pub.shape[0]
4      n=G_pub.shape[1]
5      G_pub=sympy.Matrix(G_pub)
6      message = myReadFromFile(mfile)
7      #Create a null set of codewords
8      cwords=sympy.zeros(message.shape[0],G_pub.shape[1])
9      cwordsWE=sympy.zeros(message.shape[0],G_pub.shape[1])
```

```
10
11      #Create error vector
12      errorv=sympy.zeros(1,G_pub.shape[1])
13      positions=[]
14      i=0
15      while i<t_val:
16          pos = random.randrange(G_pub.shape[1])
17          if pos not in positions :
18                  positions.append(pos)
19                  errorv[0,pos]=1
20                  i=i+1
21
22      #Encrypt the total message with public key
23      depth=message.shape[0]
24      level=0
25      while level<depth:
26          cwords[level,:]=(message[level,:] * G_pub).applyfunc(lambda x: mod(x,2))
27          level=level+1
28      #sympy.pprint(cwords)
29      #print()
30
31      #Add the errors
32      level=0
33      while level<depth:
34          cwordsWE[level,:]=(cwords[level,:] + errorv).applyfunc(lambda x: mod(x,2))
35          level=level+1
36
37      print("The error vector is: ")
38      sympy.pprint(errorv)
39      print("The encrypted message is: ")
40      sympy.pprint( cwordsWE)
41
42      #Write codeword c to file
43      writeCipher(cwordsWE,cipherFile)
44
45      dist=get_dist(cwordsWE[0,:],cwordsWE[1,:])
46
47      with open('encrypt_logs.txt','a') as f:
48          f.write('\n')
49          f.write('\n')
50          f.write(str(datetime.datetime.now()))
51          f.write('\n')
52          f.write('n,k,t='+str(n) +','+ str(k) +','+ str(t_val) +'\n'+ 'distance
                                                    between cwordsWE[0] and cwordsWE[1
                                                    ] is '+str(dist)+'\nerror is '+
                                                    sympy.pretty(errorv))
53          f.close
```

A.2.7 Decryption

```
1 def decrypt(cfile, privkey, outfile):
```

```
    S_matrix, H_matrix, P_matrix, t_val = readFromFile(privkey)
    k_val = H_matrix.shape[0]
    n_val = H_matrix.shape[1]
    S_inverse = (S_matrix**-1).applyfunc(lambda x: mod(x,2))
    P_inverse = (P_matrix**-1).applyfunc(lambda x: mod(x,2))
    cipher = myReadFromFile(cfile)
    depth=cipher.shape[0]
    #sympy.pprint(cipher)
    errors_tbit = []
    numbers = []
    for i in range(H_matrix.shape[1]):
        numbers.append(i)
    # Generate t-bit errors and syndromes
    for bits in itertools.combinations(numbers, t_val):
        et = sympy.zeros(1, H_matrix.shape[1])
        for bit in bits:
            et[bit] = 1
        st = (et * H_matrix.T).applyfunc(lambda x: mod(x,2))
        errors_tbit.append([et, st])
    message = []
    s_zeros = sympy.zeros(1,H_matrix[0])
    level=0
    while level<depth:
        print(sympy.pretty(cipher[level,:]), end=' -> ')
        mSG = (cipher[level,:] * P_inverse).applyfunc(lambda x: mod(x,2))
        s_mSG = (mSG * H_matrix.T).applyfunc(lambda x: mod(x,2))
        if not args.f:
            for errors in errors_tbit:
                if errors[1] == s_mSG:
                    recover = (mSG + errors[0]).applyfunc(lambda x: mod(x,2))
                    s_recover = (recover * H_matrix.T).applyfunc(lambda x: mod(x,2))
                    mSG = recover
        mS = mSG.extract([0], list(range(k_val, n_val)))
        m = (mS *S_inverse).applyfunc(lambda x: mod(x,2))
        if(args.v): sympy.pprint(m)
        message.append(m)
        level=level+1
    writePlain(message, outfile)
```

A.3 Information Set Decoding (ISD) Algorithms in Python

A.3.1 Plain-ISD (Prange)

```
def prange(c, H, t):
    rawH=myReadFromFile(H)
    n=rawH.shape[1]
    k= rawH.shape[1]  - rawH.shape[0]

    cword_all=myReadFromFile(c)
    cword=cword_all[0,:]
```

```
8
9       syndr=(cword*rawH.T).applyfunc(lambda x: mod(x,2))
10      alg=1
11      attempts=0
12      time.sleep(1)
13      #Algorithm inits
14      attemptsQ=0
15      while alg:
16          attempts+=1
17          print("Prange attempt number", attempts )
18          print("Creating P...")
19          #P=permutationMatrix(rawH.shape[1])
20          permutation = random.sample(range(n), n)
21          P = sympy.Matrix(n, n,
22                           lambda i, j: int((permutation[i]-j)==0))
23          HP=(rawH*P).applyfunc(lambda x: mod(x,2))
24          print("Attempting G.E...")
25          #check,primeH=Gauss_Elim(HP,k,n)
26          if HP[:,k:n].det()!=0:
27              attemptsQ+=1
28              #print('Finding Q(num.', attemptsQ,')...' )
29              try:
30                  Q=HP[:,k:n].inv_mod(2)
31
32              except ValueError:
33                  print('Unable to apply G.E, restarting...')
34                  #time.sleep(1)
35                  continue
36              #time.sleep(1)
37              primeSyndr=(syndr*Q.T).applyfunc(lambda x: mod(x,2) )
38              zeroVec=sympy.zeros(1,k)
39              primeErrorV=zeroVec.row_join(primeSyndr)
40
41              #isSyndr=(errorV*rawH.T).applyfunc(lambda x: mod(x,2))
42              if int(t)==np.count_nonzero(primeErrorV)  :
43                  errorV=(primeErrorV*P.T).applyfunc(lambda x: mod(x,2))
44                  print("Success, wt(e)=w=",t, ", error vector found ",sympy.pretty(
                                                                errorV))
45
46                  alg=0
47                  break
48              else:
49                  print('Wrong error vector found.')
50          else:
51              print('Unable to apply G.E, the random H(n-k) submatrix not invertible,
                                                        restarting...')
52      return n,k,attempts
```

A.3.2 Stern

```
1   def stern(c,H,t,p,l):
```

```
2
3       rawH=myReadFromFile(H)
4       n=rawH.shape[1]
5       k= rawH.shape[1]  - rawH.shape[0]
6       if 2*p >t or 2*p>k or p==0:
7           print('wrong p distribution or number, exiting..')
8           exit()
9
10      cword_all=myReadFromFile(c)
11      cword=cword_all[0,:]
12
13
14      syndr=(cword*rawH.T).applyfunc(lambda x: mod(x,2))
15      attempts=0
16      attemptsQ=0
17      time.sleep(1)
18      #Algorithm inits
19      alg=1
20      while alg:
21          break1=1
22          break2=1
23          attempts+=1
24          print("Stern attempt number", attempts )
25          print("Creating P...")
26          #P=permutationMatrix(n)
27          permutation = random.sample(range(n), n)
28          P = sympy.Matrix(n, n, lambda i, j: int((permutation[i]-j)==0))
29          HP=(rawH*P).applyfunc(lambda x: mod(x,2))
30          print("Attempting G.E...")
31          #check,primeH=Gauss_Elim(HP,k,n)
32          if HP[:,k:n].det()!=0:
33              attemptsQ+=1
34              #print('Finding Q(#', attemptsQ,')...' )
35              try:
36                  Q=HP[:,k:n].inv_mod(2)
37              except ValueError:
38                  print('Unable to apply G.E, restarting...')
39                  #time.sleep(1)
40                  continue
41              #time.sleep(1)
42              leftPrH= (Q*HP[:,0:k]).applyfunc(lambda x: mod(x,2))
43              kLeft,kRight=getB(k,p)
44              lenKLeft=len(kLeft[0])
45
46              A=leftPrH[0:l,0:lenKLeft]
47              B=leftPrH[0:l,lenKLeft:k]
48              C=leftPrH[l:n-k,0:lenKLeft]
49              D=leftPrH[l:n-k,lenKLeft:k]
50
51              primeSyndr=(syndr*Q.T).applyfunc(lambda x: mod(x,2))
52              pSyndrl=primeSyndr[:,0:l]
53              pSyndrR=primeSyndr[:,l:n-k]
```

```
            #errorV=sympy.zeros(1,n)
            for eL in kLeft:
                eLAT=(eL*A.T).applyfunc(lambda x: mod(x,2))
                for eR in kRight:
                    eRBT=(eR*B.T).applyfunc(lambda x: mod(x,2))
                    if pSyndrl==(eLAT+eRBT).applyfunc(lambda x: mod(x,2))  :
                        eLCTeRDT=((eL*C.T).applyfunc(lambda x: mod(x,2))+(eR*D.T).
                                                                applyfunc(lambda x
                                                                : mod(x,2))).
                                                                applyfunc(lambda x
                                                                : mod(x,2))
                        prErrR=(pSyndrR+eLCTeRDT).applyfunc(lambda x: mod(x,2))

                        lzero=sympy.zeros(1,l)
                        primeErrorV=eL.row_join(eR).row_join(lzero).row_join(prErrR)

                        #isSyndr=(errorV*rawH.T).applyfunc(lambda x: mod(x,2))
                        if  int(t)==np.count_nonzero(primeErrorV):
                            errorV=(primeErrorV*P.T).applyfunc(lambda x: mod(x,2))
                            print("Success, wt(e)=w=",t, ", error vector found ",
                                                                    sympy.pretty(
                                                                    errorV))
                            break1=0
                        else:
                                print('Wrong error vector found.')
                    if break1==0:
                        break2=0
                        break
                if break2==0:
                    alg=0
                    break
        else:
            print('Unable to apply G.E, the random H(n-k) submatrix not invertible,
                                                        restarting...')
    return n,k,attempts
```

A.3.3 BCD Algorithm

```
def ballcoll(c,H,t,p,q,l):

    #ball coll
    #Constants: n, k, w □ Z with 0 ≤ w ≤ n and 0 ≤ k ≤ n.
    #Parametek2s: p1 , p2 , q1 , q2 , k1 , k2 , ‘1 , ‘2 □ Z with 0 ≤ k1 , 0 ≤ k2 ,
                                            k = k1 + k2 , 0 ≤ p1 ≤ k1 ,
    #0 ≤ p2 ≤ k2 , 0 ≤ q1 ≤ l1 , 0 ≤ q2 ≤ ‘2 , and 0 ≤ w − p1 − p2 − q1 − q2 ≤ n −
                                            k − ‘1 − ‘2 .
    '''
    l1=random.randint(0,l)
    l2=l-l1
    isl=1
    #check if q1+q2+p+p is largek2 than t
```

```
    while isl:
        q1=random.randint(0,l1)
        q2=random.randint(0,l2)
        if q1+q2+p+p<=t:
            isl=0
    '''
    #!ball coll
    rawH=myReadFromFile(H)
    n=rawH.shape[1]
    k= rawH.shape[1]  - rawH.shape[0]
    print('n,k=',n,k)
    #p=p1+p2,q=q1+q2
    if 2*p+q >t or q>l or 2*p>k  :
        print('wrong p,q distribution or number (p+q >t , q>l , p>k , p<2), exiting
                                                    ..')
        exit()

    cword_all=myReadFromFile(c)
    cword=cword_all[0,:]

    syndr=(cword*rawH.T).applyfunc(lambda x: mod(x,2))
    attempts=0
    attemptsQ=0
    time.sleep(1)
    #Algorithm inits, 1st loop
    alg=1
    while alg:
        break1=1
        break2=1
        break3=1
        break4=1
        attempts+=1
        print("Ball Collision attempt number", attempts )
        print("Creating P...")
        #P=permutationMatrix(n)
        permutation = random.sample(range(n), n)
        P = sympy.Matrix(n, n,lambda i, j: int((permutation[i]-j)==0))
        HP=(rawH*P).applyfunc(lambda x: mod(x,2))
        print("Attempting G.E...")

        if HP[:,k:n].det()!=0:
            attemptsQ+=1
            #print('Finding Q(#', attemptsQ,')...' )
            try:
                Q=HP[:,k:n].inv_mod(2)
                #print('q is...')
            except ValueError:
                print('Unable to apply G.E, restarting...')
                #time.sleep(1)
                continue
            #time.sleep(1)
            #print('left is..')
```

```
63              leftPrH= (Q*HP[:,0:k]).applyfunc(lambda x: mod(x,2))
64              #k1,k2 are equivalent to k1,k2. The q1,q2 are to be the quantitives that
                                                        distributes the q.
65              #!!!!!!Note: k1,k2,l1,l2 should be non zero....
66              '''
67              k1,k2=getB_BallColl(k,p)
68              l1,l2=getB_BallColl(l,q)
69              '''
70              #print('k1,k2 ...is...')
71              k1,k2=getB(k,p)
72              #print('len of k1[0] is ' ,len(k1[0]))
73              if q==-1:
74                  l1,l2=getB(l,0)
75              else:
76                  l1,l2=getB(l,q)
77              lenk1=len(k1[0])
78              #sympy.pprint(k2)
79              A=leftPrH[0:l,0:lenk1]
80              #print(len(k1))
81              #εδώ ειναιτοπροβλημα
82              B=leftPrH[0:l,lenk1:k]
83              C=leftPrH[l:n-k,0:lenk1]
84              D=leftPrH[l:n-k,lenk1:k]
85
86              primeSyndr=(syndr*Q.T).applyfunc(lambda x: mod(x,2))
87              pSyndrl=primeSyndr[:,0:l]
88              pSyndrR=primeSyndr[:,l:n-k]
89
90              #find the possible l size vectors with q1+q2 errors. In other words this
                                                        is the ẽM.
91              eMlist=[]
92              for el1 in  l1:
93                  for el2 in l2:
94                      eM=el1.row_join(el2)
95                      eMlist.append(eM)
96
97              #2nd loop
98              for ek1 in k1:
99                  ek1AT=(ek1*A.T).applyfunc(lambda x: mod(x,2))
100                 for ek2 in k2:
101                     #print('Into the main loop..')
102                     #Insert the early abort
103                     #if np.count_nonzero(ek1)+np.count_nonzero(ek2)>t-q:
104                         #continue
105                     ek2BT=(ek2*B.T).applyfunc(lambda x: mod(x,2))
106                     ek1ATek2BT=(ek1AT+ek2BT).applyfunc(lambda x: mod(x,2))
107                    # print('first mult,ok')
108                     for eM in  eMlist:
109                         sL=(ek1ATek2BT+eM).applyfunc(lambda x: mod(x,2))
110                         #print('2nd add,ok')
111                         if pSyndrl==sL:
112                             #print('equality,ok')
```

```
                        ek1CT=(ek1*C.T).applyfunc(lambda x: mod(x,2))
                        #print('ek1CT')
                        ek2DT=(ek2*D.T).applyfunc(lambda x: mod(x,2))
                        #print('ek2DT')
                        ek1CTek2DT=(ek1CT+ek2DT).applyfunc(lambda x: mod(x,2))
                        #sympy.pprint(ek1CTek2DT)
                        #print(n-k-l)
                        #sympy.pprint(pSyndrR)
                        prErrR=(ek1CTek2DT+pSyndrR).applyfunc(lambda x: mod(x,2)
                                                              )
                        #print('prErrR')
                        primeErrorV=ek1.row_join(ek2).row_join(eM).row_join(
                                                              prErrR)

                        #isSyndr=(errorV*rawH.T).applyfunc(lambda x: mod(x,2))
                                                              and isSyndr==
                                                              syndr
                        if int(t)==np.count_nonzero(primeErrorV):
                            errorV=(primeErrorV*P.T).applyfunc(lambda x: mod(x,2
                                                                  ))
                            print("Success, wt(e)=w=",t, ", error vector found "
                                                                  ,sympy.
                                                                  pretty(
                                                                  errorV))
                            break4=0
                        else:
                            print('Wrong error vector found.')
                    if break4==0:
                        break3=0
                        break
                if break3==0:
                    break2=0
                    break
            if break2==0:
                alg=0
                break


    else:
        print('Unable to apply G.E, the random H(n-k) submatrix not invertible,
                                                    restarting...')

return n,k,lenk1,len(l1[0]),attempts
```

A.4 Utility Code

A.4.1 Create Keys (create_keys_cword.sh)

```
#!/bin/bash
m=$1
```

```
t=$2

./McEliece.py -g -m $m -t $t -o $m$t -v
msg='Hello'
./MatrixCodec.py -e msg -par $m$t.Hpub -v -o $m$t.binMsg
./McEliece.py -e $m$t.binMsg   -pub $m$t.pub -o $m$t.codeword
```

A.4.2 ISD Utilities

```
#!/python
import sys
import time
import numpy as np
import os
import sympy
import gzip
import pickle
import random
import itertools
import math

'''General utility functions'''

#def συνάρτησηναγραφειαπλοκείμενο
def myWriteFile(output, filename):
    with gzip.open(filename, 'wb') as f:
        f.write(pickle.dumps(output))

def mod(x, modulus):
    numer, denom = x.as_numer_denom()
    try:
        return numer*sympy.mod_inverse(denom,modulus) % modulus

    except:
        print('Error: Unable to apply modulus to matrix')

        exit()
def clear():
    os.system('clear')

def myReadFromFile(filename):
    with gzip.open(filename, 'rb') as f:
        matrix= sympy.Matrix(pickle.loads(f.read()))
    return matrix

def permutationMatrix(size):
    P=sympy.zeros(size,size)
    position=[]
    flag=False
    i=0
```

```
42      while i< size:
43          pos=np.random.randint(size)
44          if pos not in position:
45              position.append(pos)
46              i+=1
47      #print (position)
48      for i in range(size):
49          P[i,position[i]]=1
50      return P
51
52  def doLowZeros(matrix,currentRow, currentCol):
53      for row in range(currentRow+1,matrix.shape[0]):
54          if  matrix[row,currentCol] :
55              matrix=matrix.elementary_row_op(op='n->n+km',k=1,row1=row,row2=
                                                    currentRow)
56              matrix=matrix%2
57      return matrix
58  def doUpperZeros(matrix,currentRow, currentCol):
59      for row in range(currentRow-1,-1,-1):
60          if  matrix[row,currentCol] :
61              matrix=matrix.elementary_row_op(op='n->n+km',k=1,row1=row,row2=
                                                    currentRow)
62              matrix=matrix%2
63      return matrix
64  def swapRows(matrix,currentRow, currentCol):
65      for row in range(currentRow,matrix.shape[0]):
66          if matrix[row,currentCol]:
67              matrix=matrix.elementary_row_op(op='n<->m',row1=currentRow,row2=row)
68      return matrix
69  def isLower0(m,colStart,colStop):
70      falseList=[]
71      #sympy.pprint(m)
72      for r in range(m.shape[0]):
73          for c in range(m.shape[0]):
74              if c<r and m[r,c]!=0:
75                  falseList.append(999)
76      if 999 in falseList:
77          #matrix has 1 in lower triangl
78          return False
79      else:
80          #matrix has 0 in lower triangl,Bingo....
81          return True
82  def getZeroXY(matrix):
83      #matrix=M[:,colStart:colStop]
84      xList=[]
85      yList=[]
86      for i in range(matrix.shape[0]):
87          for j in range(matrix.shape[0]):
88              if i==j and matrix[i,j]==0:
89                  #print('i,j',i,j)
90                  xList.append(i)
91                  yList.append(j)
```

```
92      return xList,yList
93  #fix the matrix size into the functions
94  def fixDiagonal(M,colStart,colStop):
95      matrix=M[:,colStart:colStop]
96      sampleDiag=sympy.ones(1,colStop-colStart)
97      #while
98      xL,yL=getZeroXY(matrix)
99      #print(xL,yL)
100     while xL:
101         x=xL.pop()
102         y=yL.pop()
103         #for r in range(matrix.shape[0]):
104         r=np.random.randint(matrix.shape[0])
105         #print(matrix[r,y])
106         if matrix[r,y]:
107             M=M.elementary_row_op(op='n->n+km',k=1,row1=x,row2=r)
108             M=M%2
109
110     if sampleDiag!=matrix.diagonal():
111         return False, M
112     else:
113         return True,M
114 '''Gauss elimination with row operations'''
115 def Gauss_Elim(matrix,colStart, colStop):
116     flag1=0
117     flag2=0
118     while not flag1 and not flag2:
119         #sympy.pprint(matrix)
120         row=-1
121         for col in range(colStart,colStop):
122             row+=1
123             if matrix[row,col]:
124                 matrix=doLowZeros(matrix,row,col)
125             else:
126                 matrix=swapRows(matrix,row,col)
127         flag2,matrix=fixDiagonal(matrix,colStart,colStop)
128         flag1=isLower0(matrix[:,colStart:colStop],colStart,colStop)
129     row=matrix.shape[0]
130     for col in range(colStop-1,colStart, -1):
131         row-=1
132         #print('row,col', row,col)
133         matrix=doUpperZeros(matrix,row,col)
134
135     if matrix[:,colStart:colStop]==sympy.eye(colStop-colStart):
136         return 1,matrix
137     else:
138         return 0,matrix
139
140 def randVec(size,ones):
141     vec=sympy.zeros(1,size)
142     posList=[]
143     i=0
```

```
144     while i < ones:
145         pos=random.randint(0,size-1)
146         if pos not in posList:
147             posList.append(pos)
148             i+=1
149     for j in posList:
150         vec[0,j]=1
151     return vec
152 
153 
154 def getPossibleVectors(size,ones):
155     veclist=[]
156     over=(math.factorial(ones)*math.factorial(size-ones))
157     listSize=int(math.factorial(size)/over)
158     isSize=0
159     #print(listSize)
160     while isSize<listSize:
161         shuffled=randVec(size,ones)
162         if shuffled  in veclist:
163             continue
164         else:
165             veclist.append(shuffled)
166             isSize+=1
167     return veclist
168 
169 def getB_bc(k,p):
170     #Seperate randomly the k to k1,k2
171     if k%2==0:
172         k1=random.randint(int(k/4),int(k/2))
173     else:
174         k1=random.randint(int(k-1/4),int(k-1/2))
175 
176 
177     #k1=int(2*k/3)
178     #k1=3
179     k2=k-k1
180     p1=1
181     p2=p-1
182     return getPossibleVectors(k1,p1), getPossibleVectors(k2,p2)
183 
184 
185 
186 def getB(k,p):
187     #if k is odd seperates the k
188     if k%2!=0:
189         floor=int((k-1)/2)
190         ceil=floor+1
191         #floorList=list(itertools.product([int(0), int(1)], repeat=floor))
192         #ceilList=list(itertools.product([int(0), int(1)], repeat=ceil))
193         return getPossibleVectors(floor,p), getPossibleVectors(ceil,p)
194     #k is not odd
195     else:
```

```
196
197         list=getPossibleVectors(int(k/2),p)
198         #left_right=list(itertools.product([int(0), int(1)], repeat=int(k/2)))
199         return list ,list
```

A.4.3 McEliece Utilities

```
1  #!/usr/bin/python3
2  import argparse
3  import gzip
4  import pickle
5  import sympy
6  import random
7
8  def mod(x, modulus):
9      numer, denom = x.as_numer_denom()
10     try:
11         return numer*sympy.mod_inverse(denom,modulus) % modulus
12
13     except:
14         print('Error: Unable to apply modulus to matrix')
15
16         exit()
17 def get_dist(a_matrix, b_matrix):
18     diff_matrix = (a_matrix + b_matrix).applyfunc(lambda x: mod(x,2))
19     dist = (diff_matrix * sympy.Matrix([1] * diff_matrix.shape[1]))[0]
20     return dist
21
22 def writeCipher(output, filename):
23     with gzip.open(filename , 'wb') as f:
24         f.write(pickle.dumps(output))
25 def writePlain(output, filename):
26     with gzip.open(filename , 'wb') as f:
27         f.write(pickle.dumps(output))
28 def readFromFile(filename):
29     with gzip.open(filename, 'rb') as f:
30         contents = pickle.loads(f.read())
31     return contents
32 def writeToFile(filename):
33     with gzip.open(filename, 'rb') as f:
34         contents = pickle.loads(f.read())
35     return contents
36 def writeKeys(Gen_matrix,G_matrix, t_val, S_matrix, H_matrix,H_pub, P_matrix,
                                          filename):
37     with gzip.open(filename + '.pub', 'wb') as f:
38         f.write(pickle.dumps([G_matrix, t_val]))
39     with gzip.open(filename + '.priv', 'wb') as f:
40         f.write(pickle.dumps([S_matrix, H_matrix, P_matrix, t_val]))
41     with gzip.open(filename + '.parity', 'wb') as f:
42         f.write(pickle.dumps([H_matrix]))
43     with gzip.open(filename + '.gen', 'wb') as f:
```

```
        f.write(pickle.dumps([Gen_matrix]))
    with gzip.open(filename + '.Hpub', 'wb') as f:
        f.write(pickle.dumps([H_pub]))


parser = argparse.ArgumentParser()
parser.add_argument("-v", help="Enable verbose mode", action="store_true")
parser.add_argument("-vv", help="Enable very verbose mode",
          action="store_true")
parser.add_argument("-g", help="Generate key pairs", action="store_true")
parser.add_argument("-m", type=int, help="Generate key pairs")
parser.add_argument("-t", type=int, help="Generate key pairs")
parser.add_argument("-o", type=str, help="Output file (always needed)")
parser.add_argument("-e", type=str,
          help="File with data in matrices to encrypt")
parser.add_argument("-d", type=str,
          help="File with data in matrices to decrypt")
parser.add_argument("-pub", type=str, help="Key to encrypt with")
parser.add_argument("-priv", type=str, help="Key to decrypt with")
parser.add_argument("-f",help="Encrypt without errors in ciphertext", action="
                                        store_true")
''' my adds'''
parser.add_argument("-x", type=str, help="Encoded msg without errors")
parser.add_argument("-par", type=str, help="Parity matrix")
parser.add_argument("-cw", type=str,  help="Codeword")
parser.add_argument("-er", type=str,  help="error vector")
parser.add_argument("-pErr", type=int,  help="errors in section A(stern)")
parser.add_argument("-l", type=int, help="size of dimension l(stern)")
parser.add_argument("-q", type=int, help="errors of dimension l(ballcoll)")
args = parser.parse_args()
```

A.4.4 Codec Functions

```
#!/usr/bin/python3
import argparse
import sympy
import gzip
import pickle
from sympy.combinatorics import Permutation
from McElieceUtil import *

def encode(text, parity):
    h=readFromFile(parity)
    H=sympy.Matrix(h)
    k_val=H.shape[1]-H.shape[0]
    output = []
    current = sympy.Matrix()
    for c in text:
        ascii_val = ord(c)
        for i in range(6, -1, -1):
            if current.shape[1] == k_val:
```

```
                output.append(current)
                current = sympy.Matrix()
            bit = pow(2, i)
            if ascii_val >= bit:
                ascii_val -= bit
                if current.shape[1] == 0:
                    current = sympy.Matrix([1])
                else:
                    current = current.col_insert(current.shape[1],
                                sympy.Matrix([1]))
            else:
                if current.shape[1] == 0:
                    current = sympy.Matrix([0])
                else:
                    current = current.col_insert(current.shape[1],
                                sympy.Matrix([0]))
    nonempty = current.shape[1]
    final = sympy.Matrix([1] * nonempty).T
    if final.shape[1] == 0:
        final = sympy.Matrix([0] * nonempty).T
    elif not nonempty == k_val:
        current = current.row_join(sympy.Matrix([0] * (k_val - nonempty)).T)
        final = final.row_join(sympy.Matrix([0] * (k_val - nonempty)).T)
        mix = Permutation.random(final.shape[1]).array_form
        randomize = []
        for i in mix:
            randomize.append(final[i])
        final = sympy.Matrix(randomize).T
    output.append(current)
    output.append(final)
    if args.v:
        print('The encoded text is:')
        for o in output:
            sympy.pprint(o)
    if args.o:
        with gzip.open(args.o, 'wb') as f:
            f.write(pickle.dumps(output))
def decode(filename):
    with gzip.open(filename, 'rb') as f:
        input = pickle.loads(f.read())
    pad = input[-1]
    used_bits = (pad * sympy.Matrix([1] * pad.shape[1]))[0]
    input = input[0:-1]
    for i in range(pad.shape[1] - used_bits):
        input[-1].col_del(input[-1].shape[1] - 1)
    bit_count = 6
    ascii_val = 0
    output = ''
    for m in input:
        for i in range(m.shape[1]):
            ascii_val += m[i] * pow(2, bit_count)
            bit_count -= 1
```

```
71              if bit_count < 0:
72                  bit_count = 6
73                  output = output + chr(ascii_val)
74                  ascii_val = 0
75      if args.v: print(output)
76  parser = argparse.ArgumentParser()
77  parser.add_argument("-o", type=str, help="File to store output")
78  parser.add_argument("-e", type=str,
79            help="String to be encoded into matrices")
80  parser.add_argument("-d", type=str,
81            help="File with matrices to be decoded to ASCII")
82  parser.add_argument("-k", type=int, help="Length of each matrix")
83  parser.add_argument("-v", help="Enable verbose mode", action="store_true")
84
85  parser.add_argument("-par", type=str, help="Parity matrix")
86
87  args = parser.parse_args()
88  if args.e and args.par:
89      encode(args.e, args.par)
90  elif args.d:
91      decode(args.d)
92  else:
93      print(parser.format_help())
```

A.5 Readme

The source code of this attempt is in https://github.com/konnnGit/mceliece-isd/tree/final. The steps one may follow to deploy the overall application are:

1. Construct a McEliece cryptosystem based on the Galois Field parameters m and t. For this paradigm $m = 7$ and $t = 9$.

   ```
   McEliece.py -g -m 7 -t 9 -o 79 -v
   ```

   Note: The flag -o is used to define the name of the necessary binary files which contain the keys, etc. The m, t parameters are part of the file names. See manual pages ?? for more information.

   Note: There is a restriction while creating keys. It is recommended not to use big values for m and t because it stalls the create process.

2. Encode a plain text message into a binary message using the public version of the Parity Check Matrix H ($Hpub = HP$).

   ```
   MatrixCodec.py -e 'Hello World' -par 79.Hpub -v -o 79.binMsg
   ```

3. Encrypt the binary message into a codeword using the McEliece public key.

   ```
   McEliece.py -e 79.binMsg   -pub 79.pub -o 79.codeword
   ```

Note: One may use the included bash script

```
./create_keys_cword.sh 7 9
```

in order to execute the steps 1,2 and 3 simultaneously. In that case, the message to be sent is included in the script. Editing it one can be change it.

4. Attack the cipher having three (3) options:

   a) Prange's algorithm

   ```
   prange.py -c 5 -m 7 -t 9
   ```

   Note: In order to run the algorithms in background one may use the *nohup* command, like

   ```
   nohup prange.py -c 5 -m 7 -t 9 &
   ```

   b) Stern's algorithm

   ```
   stern.py -c 35 -m 7 -t 9 -p 1 -l 2
   ```

   Note: The flag *-p* concerns the error bits in each $k/2$ coordinates.

   Note: The flag *-c* indicates how many times will be repeated the algorithm.

   c) Ball-collision-Decoding algorithm

   ```
   ball-coll.py -c 35 -m 7 -t 9 -p1 2 -p2 1 -q1 -1
       -q2 -1 -l 2
   ```

   Note: In order to pass zero errors at any of -q1, -q2 coordinates, this must declared with a negative value.

5. Decrypt the cipher. In case of decrypting, the flag -d must be selected. We coose the correspoding codeword, the private key and define the output palin text name.

   ```
   McEliece.py -d 79.codeword -priv 79.priv -o 79.plain
   ```

6. Decode the decrypted binary cipher text into the initial plain text.

   ```
   MatrixCodec.py -d 79.plain
   ```

Restrictions:

Bibliography

[1] Rivest, Shamir and Adleman. "A method for obtaining digital signatures and public-key cryptosystems".Communications of the ACM, 1978. pp. 120-126. http://people.csail.mit.edu/rivest/Rsapaper.pdf

[2] Shor, Peter W. "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer". https://doi.org/10.1137/S0036144598347011

[3] Diffie, Whitfield; Hellman, Martin (November 1976). "New Directions in Cryptography" (PDF). IEEE Transactions on Information Theory. IT-22 (6): 644654. CiteSeerX 10.1.1.37.9720. doi:10.1109/tit.1976.1055638.

[4] "FIPS PUB 197: The official Advanced Encryption Standard". Computer Security Resource Center. National Institute of Standards and Technology.

[5] "NCUA letter to credit unions". National Credit Union Administration. July 2004.

[6] Menezes, A.J.; van Oorschot, P.C.; Vanstone, S.A. (1997). Handbook of Applied Cryptography. ISBN 978-0-8493-8523-0.

[7] R,J McEliece, A public-key cryptosystem based on algebraic coding theory, 1978.

[8] V. D. Goppa,A New Class of Linear Correcting Codes, Problems of Information Transmission6 (1970), pp. 207212.

[9] Kumari, Saru, et al. "A secure authentication scheme based on elliptic curve cryptography for IoT and cloud servers." The Journal of Supercomputing 74.12 (2018): 6428-6453.

[10] Panayiota T. Smyrli 1, Nicholas Kolokotronis, Konstantinos Limniotis MULTILAYER CONSTRUCTIONS OF ISD ALGORITHMS FOR SOLVING THE CSD PROBLEM, 2020

[11] Zhe Li, Chaoping Xing,Sze Ling Yeo, Reducing the Key Size of McEliece Cryptosystem from Automorphism-induced Goppa Codes via Permutations,2019

[12] Valentijn, Ashley, "Goppa Codes and Their Use in the McEliece Cryptosystems" (2015). Syracuse University Honors Program Capstone Projects. 845. *https* : *//surface.syr.edu/honors*$_c$*apstone/*845

[13] S. Mrdovic and B. Perunicic, "Kerckhoffs' principle for intrusion detection," Networks 2008 - The 13th International Telecommunications Network Strategy and Planning Symposium, 2008, pp. 1-8, doi: 10.1109/NETWKS.2008.6231360.

[14] Stephane Beauregard, Circuit for Shor's algorithm using $2n + 3$ qubits https://arxiv.org/abs/quant-ph/0205095

[15] American Journal of Physics 73, 521 (2005); doi: 10.1119/1.1891170

[16] Verdu, S., 1998. Fifty years of Shannon theory. IEEE Transactions on information theory, 44(6), pp.2057-2078.

[17] R. McEliece and L. Swanson, "On the decoder error probability for Reed - Solomon codes (Corresp.)," in IEEE Transactions on Information Theory, vol. 32, no. 5, pp. 701-703, September 1986, doi: 10.1109/TIT.1986.1057212.

[18] Stern, Jaques. A Method For Finding Codewords of Small Weight. Coding Theory and Applications, 388: 106-133, 1989

[19] R. Agrawal, L. Bu and M. A. Kinsy, "Quantum-Proof Lightweight McEliece Cryptosystem Co-processor Design," 2020 IEEE 38th International Conference on Computer Design (ICCD), 2020, pp. 73-79, doi: 10.1109/ICCD50377.2020.00029.

[20] Grover, Lov K. "A fast quantum mechanical algorithm for database search." In Proceedings of the twenty-eighth annual ACM symposium on Theory of computing, pp. 212-219. 1996.

[21] Grover, Lov K. "Quantum mechanics helps in searching for a needle in a haystack." Physical review letters 79, no. 2 (1997): 325.

[22] Bernstein, Daniel J. "Grover vs. mceliece." In International Workshop on Post-Quantum Cryptography, pp. 73-80. Springer, Berlin, Heidelberg, 2010.

[23] Daniel J. Bernstein, Tanja Lange, Christiane Peters, Attacking and defending the McEliece cryptosystem, in [9] (2008), 3146. URL: http://eprint.iacr.org/2008/318. Citations in this document: ğ2, ğ2.

[24] E. Prange, The use of information sets in decoding cyclic codes, IRE Transactions on Information Theory, Vol.8(5), 1962.

[25] J. Stern. A method for finding codewords of small weight. In G. D. Cohen and J. Wolfmann, editors, Coding Theory and Applications, volume 388 of Lecture Notes in Computer Science, pages 106113. Springer, 1988.

[26] Meurer, Alexander. "A coding-theoretic approach to cryptanalysis." PhD diss., Verlag nicht ermittelbar, 2013.

[27] May, Alexander, Alexander Meurer, and Enrico Thomae. "Decoding random linear codes in $\tilde{O}(2^{0.054n})$." In International Conference on the Theory and Application of Cryptology and Information Security, pp. 107-124. Springer, Berlin, Heidelberg, 2011.

[28] Bernstein, Daniel J., Tanja Lange, and Christiane Peters. "Smaller decoding exponents: ball-collision decoding." In Annual Cryptology Conference, pp. 743-760. Springer, Berlin, Heidelberg, 2011.

[29] Shafique, Kinza, Bilal A. Khawaja, Farah Sabir, Sameer Qazi, and Muhammad Mustaqim. "Internet of things (IoT) for next-generation smart systems: A review of current challenges, future trends and prospects for emerging 5G-IoT scenarios." Ieee Access 8 (2020): 23022-23040.

[30] Cheng, Chi, Rongxing Lu, Albrecht Petzoldt, and Tsuyoshi Takagi. "Securing the Internet of Things in a quantum world." IEEE Communications Magazine 55, no. 2 (2017): 116-120.

[31] https://iot-analytics.com/iot-market-size/

[32] Bu, Lake, Rashmi Agrawal, Hai Cheng, and Michel A. Kinsy. "A lightweight McEliece cryptosystem co-processor design." arXiv preprint arXiv:1903.03733 (2019).

[33] Eisenbarth, Thomas, Tim Güneysu, Stefan Heyse, and Christof Paar. "MicroEliece: McEliece for embedded devices." In International Workshop on Cryptographic Hardware and Embedded Systems, pp. 49-64. Springer, Berlin, Heidelberg, 2009.

[34] M. Y. Hsiao, D. C. Bossen and R. T. Chien, "Orthogonal Latin Square Codes," in IBM Journal of Research and Development, vol. 14, no. 4, pp. 390-394, July 1970, doi: 10.1147/rd.144.0390.