

# Secure Messaging System Based on McEliece

## Encryption Scheme



**MCS**

By

Humera Javed

00000319724

Submitted to the faculty of Information Security Department, Military College of Signals,  
National University of Sciences and Technology, Rawalpindi in partial fulfillment of the  
requirements for the degree of MS in Information Security

Jun 2023

**THESIS ACCEPTANCE CERTIFICATE**

Certified that final copy of MS Thesis written by **Ms Humera Javed**, Registration No. **00000319724**, of **Military College of Signals** has been vetted by undersigned, found complete in all respects as per NUST Statutes/Regulations/MS Policy, is free of plagiarism, errors, and mistakes and is accepted as partial fulfillment for award of MS degree. It is further certified that necessary amendments as pointed out by GEC members and local evaluators of the scholar have also been incorporated in the said thesis.

Signature: \_\_\_\_\_ 

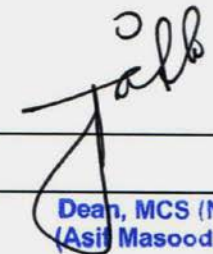
Name of Supervisor Brig, Abdul Ghafoor, PhD

Date: \_\_\_\_\_

Signature (HOD): \_\_\_\_\_ 

Date: 21-8-2023

Information Security  
Military College of Sigs

Signature (Dean/Principal) \_\_\_\_\_ 

Date: 21/8/23

Brig  
Dean, MCS (NUST)  
(Asif Masood, PhD)

## Table of Contents

Certificate	6
Declaration	7
Dedication	8
Abstract	9
Acknowledgments	10
Introduction	11
1.1 Overview	11
1.2 Motivation and Problem Statement	11
1.3 Research Objective	12
1.4 Scope of Research	12
1.5 Significance of Research	13
1.6 Thesis Organization	14
Literature Review	15
2.1 Limitation of the existing literature	20
Post Quantum Cryptography	21
3.1 Introduction	21
3.2 Explanatory Chart	22
3.3 Algorithms	22
3.3.1 Lattice-based cryptography	22
3.3.2 Multivariate cryptography	22
3.3.3 Hash-based cryptography	23
3.3.4 Code-based cryptography	23
3.3.5 Super singular elliptic curve isogeny cryptography	23
3.3.6 Symmetric key quantum resistance	24
3.4 Key Attributes of PQC Algorithms	25
3.6 Security reductions	28
3.6.1 Lattice-based cryptography – Ring-LWE Signature	28
3.6.2 Lattice-based cryptography – NTRU, BLISS	28
3.6.3 Multivariate cryptography – Unbalanced Oil and Vinegar	28
3.6.4 Hash-based cryptography – Merkle signature scheme	29
3.6.5 Code-based cryptography – McEliece	29
3.6.6 Code-based cryptography – RLCE	29
3.6.7 Super singular elliptic curve isogeny cryptography	29

3.7	Comparison	29
	McEliece cryptosystem	31
4.1	Introduction	31
4.2	Part I: Basic Terminology	33
4.1.1	Cryptology	33
4.1.2	Error Correcting Codes	34
4.1.3	Fields	34
4.1.4	Binary	37
4.1.5	Hamming Distance	37
4.1.6	Linear Codes	38
4.3	Goppa Codes	38
4.2.1	Parameters	39
4.2.2	Binary Goppa Codes	39
4.2.3	Parity Check Matrix	39
4.2.4	Encoding	40
4.2.5	Irreducible Binary Goppa Code Example	41
4.2.6	Error Correction	43
4.2.7	Decoding	45
4.4	The McEliece Cryptosystem with Example	45
4.3.1	Example	46
	Application Implementation and Results	49
5.1	Application GUI	49
5.2	Key Generation:	50
5.3	Keys Generation	65
5.4	Encryption and Decryption Process	71
5.3.1	Private Key	72
5.3.2	Public Key	73
	Discussion and Analysis	78
6.1	Creation of Parameters	79
6.1.1	Finite Field:	79
6.1.2	Goppa Polynomial:	80
6.2	Number of Random Vectors tested while decryption	83
6.3	Initial Parameters vs. Key length	83

6.4	Encryption and Decryption Time	84
	References	87

## Certificate

This is to certify that NS **Humera Javed** Student of **MSIS-18** Course Reg. No. **00000319724** has completed his MS Thesis title “**Secure Messaging System Based on McEliece Encryption Scheme**” under my supervision. I have reviewed his final thesis copy and am satisfied with his work.

Dated: \_\_\_May 2023

---

Thesis Supervisor  
**(Brig Dr. Abdul Ghafoor)**

## **Declaration**

I hereby declare that no portion of the work presented in this thesis has been submitted in support of another award or qualification either at this institution or elsewhere.

---

**Humera Javed**

## **Dedication**

I dedicate this thesis to Almighty Allah, my source of inspiration, wisdom, knowledge, blessing, and understanding.

My thesis is also dedicated to my families, who have been very supportive during the duration of my endeavor.

This thesis is also dedicated to the people who inspired me to work hard and never give up on a goal I set for myself.



## Abstract

The security of important applications of the internet such as e-commerce, banking, and eHealth is furnished by public-key cryptographic systems (or cryptosystems, for short). Underlying hard problems of presently used public cryptosystems are factoring and computing discrete logarithms which are unable to solve by conventional computers. Main disadvantage of these systems is that they are not resistant against quantum computers. It is anticipated that during the next 10–20 years, quantum computers will be ready for widespread use. Code based encryption systems are one of the cryptosystems that are capable to resist quantum computing and hence provides an area in Post-Quantum Cryptography. McEliece cryptosystem which is also a code-based cryptosystem is one of the few that is expected to withstand future assaults from powerful computers. Despite of several attempts by the crypto community, the McEliece system has not been cracked to this day.

In recent years, internet-based messaging applications have been widely used as they make it easy to communicate and connect with people around the world. There are many chat apps that offer secure messaging services by using conventional encryption schemes however, they are not quantum resistive. So, need of the time is to implement the messaging System based on McEliece encryption scheme that is prone to Quantum attacks. The McEliece cryptographic system is one of the suitable options to ensure secure communications over the Internet when quantum computers become practical.

By using McEliece public key encryption system we have proposed/developed Quantum-resistant communications system/ app. It is a desktop app that can be used in computers for secure and Quantum resistive environment. The final product that we will be providing to the two users is just an exe file that can be installed in any computer and shortcut as well as the icon will be visible on the desktop. As the underlying encryption scheme used in the app is McEliece encryption scheme so it is supposed to be resistive against cyber-attacks generated by quantum computers. Moreover, this application may be integrated with financial systems to facilitate secure transactions. The secure nature of app can prevent unauthorized access to sensitive financial information, making it harder for hackers to intercept or manipulate transactions. This can enhance the security and integrity of digital payments, banking, and other financial services. It can also be used in Government and military organizations for communication. Governments and military organizations deal with highly sensitive information, and secure communication is crucial for their operations. This app can ensure that their messages remain confidential and resistant to interception by adversaries.

## Acknowledgments

I express my deepest gratitude to my supervisor, Brig. Dr. Abdul Ghafoor, for his countless hours in reflecting, reading, encouraging, guidance and continuous support throughout this journey. It was a great privilege and honor for me to work and study under his supervision. His vision, dynamism, prompt response and valuable suggestions make it possible to complete this tedious task in due course of time. His suggestions helped me to improve my problem-solving capabilities and research skills.

I would like to extend my gratitude to co-supervisor Dr. Shahzaib Tahir Butt and committee members Dr. Fawad Khan and Maj Bilal Ahmed for their support and recommendations, which improved this research study.

For the chance to hone my research abilities and have a positive impact on the world, I am very grateful to the College of Signals and National University of Science and Technology.

I am also very thankful to the organization and individuals who participated in this research study directly or indirectly to implement and test the proposed idea.

Finally, I would want to express my appreciation to everyone who has helped me in any way with this research project, whether it be via providing me with information or advice.

Thank you all.

## **Introduction**

### **1.1 Overview**

The security of today's traditional cryptosystems is increasingly at risk as quantum computers become closer to commercialization and can solve the issues that these systems rely on, such as integer factorization and discrete logarithm. Since quantum computers can effectively solve certain problems that classical computers cannot, they will eventually be able to crack the security protocols now in use. We need to create cryptographic systems that can function in a post-quantum world so that we are ready for the day when quantum computers become a realistic reality. In order to achieve this objective, it is necessary to investigate other computing problems that are difficult for both conventional and quantum computers. “The National Institute of Standards and Technology (NIST) in the United States is now running a competition to standardize a public-key encryption system suitable for use once quantum computers have been built.

### **1.2 Motivation and Problem Statement**

Quantum computers, which are still in their infancy but have the potential to greatly boost the speed with which specific problems like the ones listed above are addressed by conventional computers, are now the subject of intense research and development. Within the next decade to two decades, quantum computers may find widespread use [31]. If they materialized now, they would instantly make vast sections of the internet unsafe.

For these reasons, in November 2017, NIST held a competition to determine which, if any, quantum-resistant public-key cryptosystems should be adopted as the industry standard. McEliece crypto system was one of the finalists of NIST’s Post Quantum Cryptography Standardization Process round 02. Other Candidates in second round of NIST along with McEliece cryptosystem were as listed below:-

- BIKE
- CRYSTALS-KYBER
- FrodoKEM
- HQC

- LAC
- LEDAcrypt (merger of LEDAkem/LEDApkc)
- NewHope
- NTRU (merger of NTRUEncrypt/NTRU-HRSS-KEM)
- NTRU Prime
- NTS-KEM
- ROLLO (merger of LAKE/LOCKER/Ouroboros-R)
- Round5 (merger of Hila5/Round2)
- RQC
- SABER
- SIKE
- Three Bears

In principle, the advent of quantum computers makes obsolete long-standing public key cryptosystems predicated on the challenge of computing logarithms over finite fields. The McEliece system is one of the few that is expected to withstand assaults from powerful computers in the future, despite several efforts by the crypto community to break it. Because of the vulnerability of our current communications infrastructure to Quantum assaults, the adoption of a Quantum-Resistant Crypto System has become an urgent need. When quantum computers become commonplace, we'll need to employ the McEliece cryptography scheme to keep Internet connections safe.

### **1.3 Research Objective**

Research objective of the thesis is as follow:-

- In depth analysis of McEliece Cryptosystem in comparison with available Quantum resistive algorithms
- Implementation of McEliece Cryptosystem based on Goppa Codes
- Implementation of Secure Quantum Resistive messaging application
- Analysis of Quantum Resistive Secure Communication messaging Solution

### **1.4 Scope of Research**

In this thesis we developed a Messaging application that uses McEliece encryption. We can use this application anywhere in between two clients that have insecure media or channel. Most of the

application now a days use conventional encryption techniques, this is out of the box solution for secure messaging application. Specifically on wireless Medias where most of the messaging applications cannot be used, this solution after implementation can be tested.

Some of the advantages of this research will be:

- Develop an understanding of McEliece cryptosystem
- Develop an understanding asymmetric messaging architecture between two clients
- Design and implementation of McEliece cryptosystem using Goppa Codes for secure encrypted messaging
- Paving a way for future work to include other features like Audio, Videos, and images encryption using McEliece cryptosystem

### **1.5 Significance of Research**

Secure communication between two parties, even in the presence of an eavesdropper, is one of the main goals of cryptography. An area of cryptography known as "post-quantum cryptography" focuses on classical cryptographic methods that are expected to be secure against quantum assaults. In this thesis, we will examine the use of a supposedly Quantum-resistant code-based encryption system in a secure communication application. In 1978, after the RSA cryptosystem had already been introduced, Robert McEliece suggested it. The security cryptosystem is predicated on issues that are thought to be intractable by quantum computers, making it resistant to assaults that can crack it in polynomial time. To protect sensitive information, modern cryptosystems like RSA use mathematical challenges like factoring large integers and solving the discrete logarithm problem. Integer factorization and the discrete logarithm problem in a finite field may be solved in polynomial time with the help of a technique discovered by Peter W. Shor [7] in the late 1990s. While quantum computers only exist in theory at the moment, it is believed that a working prototype might be constructed in the near future [4]. NIST has just released a document [8] in which they state:

*“It is unclear when scalable quantum computers will be available. However, [...] it is likely that a quantum computer capable of breaking 2000-bit RSA in a matter of hours could be built by 2030 for a budget of about a billion dollars.”*

This gives rise to a justifiable worry that widely used cryptosystems like RSA may be cracked over the next few years. Code-based cryptography is one such alternate approach in the scenario when quantum computers will be practically in use. For the same reason, we have implemented the

McEliece cryptosystem in a closed-source messaging app that is meant to be immune to quantum assaults.

Our study's focus is on "Development of Quantum Resistive Messaging Solution," however the underlying architecture may be used in following sectors as well.

- Wireless Communication
- May be integrated with financial systems to facilitate secure transactions
- E-commerce and Online Retail-Secure communication between buyers and sellers
- Can be used in Govt and Military organizations to ensure the confidentiality of messages and resistance to interception by adversaries.

As scope of this research we are now making a secure chat application that can be deployed at Computers and end to end encrypted messages can be forwarded which are encrypted based on McEliece algorithms. As a future work our work can be extended to make this application workable on mobile phone or on even hardware to achieve desired results.

## **1.6 Thesis Organization**

The thesis is structured as follows:

- **Chapter 1** covers the introduction part of the thesis that enlightens/highlights the problem statement, research objectives, thesis scope, and its contribution.
- **Chapter 2** is dedicated to essential literature review on the subject topic that helps us in writing and developing the Quantum resistive application.
- **Chapter 3** covers the different Quantum resistive techniques and algorithms that are developed over the period of time and also lay down basis on choice of code based McEliece Cryptosystem.
- **Chapter 4** covers the McEliece Crypto system basics terminologies, Public and Private Key generation algorithm along with the encryption and decryption process.
- **Chapter 5** explains the implementation of the secure Quantum resistive messaging application.
- **Chapter 6** presents the analysis of the developed application.
- **Chapter 7** concludes the reporting part of the research/ thesis and proposes a future research direction.

## **Literature Review**

In 1978, R. J. McEliece introduced the McEliece Encryption technique based on the Binary Goppa code, which appears highly safe while still allowing extraordinarily quick communication rates [9]. This cryptosystem is well suited for deployment in a distributed communication network, such as those expected to be used by NASA to disseminate data gathered from orbit.

In 1986, Niederreiter introduced a cryptosystem that uses a public key with fewer bits than ChorRivest but achieves a greater information rate [10]. This secure network was developed using Algebraic Coding Theory.

In 1988, Leon provided an approach for determining the smallest possible size of massive error-correcting codes. Stern provided a method for locating lightweight code words, while Lee made a security remark about the McEliece public key cryptosystem. The matrix  $B$  may be determined using  $O(s^4 + sN)$   $F_q$  arithmetic operations, thanks to an approach reported by Sidelnikov et al in 1992 [11], which exploited a weakness in [9], [10]. This technique is used to show the vulnerability of public-key cryptosystems like these.

In the same year, 1994, Sidelnikov et al proposed a further enhancement to the cryptosystem [12] described in [9] and [10] and offered some supporting data. They also looked at how difficult it would be to break the original and updated versions of the encryption scheme. They concluded that the latter had a significantly higher level of security for numbers  $N$  larger than or equal to 1024.

A year and a half later, in 1996, Janwa et al examined important versions of the McEliece encryption method employing a new and bigger class of  $q$ -ary A-G (Algebraic Geometric) Goppa codes [13]. In 1998, Sendrier investigated code recovery by constructing a linear code  $C$  from its generating matrix to create a concatenated pattern [14].

The McEliece cryptography approach was strengthened by Loidreau et al in 2000 without needing a larger public key [7]. They could do this by generating decipherable patterns of large-weight errors utilizing certain regions of the automorphism groups of the codes.

In 2000, Canteaut et al issued a rebuttal to [9]. This technique is a novel probabilistic method for identifying the low-weight words in any large linear code. Public key cryptography with key sizes of fewer than 4000 bits was suggested by Berger et al in 2005 [16]. Also, they demonstrated how to conceal the underlying code structure while exploiting the features of subcodes to shrink the public key size.

In addition, Minder et al demonstrated in 2007 how the Sidelnikov cryptosystem might be compromised based on its design [6]. A private key is generated from a public key in this attack. If the parameters efficiently sample the code word with the least weight, then Reed-Muller is an effective code. The execution time of the code is sub-exponential in this example. McEliece, a cryptosystem built on low-genus curves, is the primary target of this attack.

For their 2008 study, Baldi et al settled on a group of QC-LDPC Codes (Quasi-Cyclic Low-Density Parity Check Codes) [17]. Due to its reliance on Goppa codes, the initial McEliece cryptosystem suffered from large key sizes and sluggish transmission rates. However, these issues were circumvented. Codes are generated utilizing a novel technique based on Random Difference Families that allows for generating a huge number of sets of mutually-indistinguishable codes. Comprehensive cryptanalysis was built to confirm the security level attainable via a selection of system characteristics. To address the major issues identified as potentially harmful attacks, they refined a QC-LDPC code implementation of the McEliece cryptographic system. Baldi made a few adjustments throughout the next year. It was proposed in 2009 that a public key may be shrunk by creating quasi-cyclic codes over  $F_{28}$ .

A cryptanalysis based on QC (Quasi Cyclic) codes and their variants was published in 2010; this method ultimately allowed the McEliece encryption scheme to be cracked. Berger and Loidreau developed Subcodes for generalized Reed Solomon codes. A novel structural attack on the McEliece/Niederreiter public key cryptosystem was developed in the same year by Wieschebrink et al [18]. Thus, for almost all practicable parameter selections, the private key can be rebuilt with high confidence in polynomial time.



In 2012, two variants of the McEliece cryptosystem were developed, one based on MDPC (Moderate Density Parity-Check) codes and another on QC-MDPC (Quasi Cyclic MDPC) codes [6]. Everything, from generating keys to encrypting and decrypting data, is simplified in this new edition. That year, a novel kind of public key encryption using convolutional codes called McEliece was also introduced. Important to this design are random-generated parity checks. They hinder one's ability to mount a structural assault.

Coureur et al demonstrated that the A-G-coded McEliece public key encryption scheme could be cracked in under two years using a polynomial-time attack [5]. A public-key decoding technique can be found for the first time, even for codes constructed on high-genus curves. Because of this, their assaults are very effective compared to their predecessors.

In 2014, Iltis et al presented hexi polynomial codes, hexi Maximum Rank Distance codes, hexi Rank Distance codes, hexi wild Goppa codes, and hexi Goppa codes [19]. These new McEliece codes were used to develop a public-key cryptosystem called Hexi McEliece. The temporal complexity of this newly developed cryptosystem is lower, and its error-correcting capability is higher, making it more practical for widespread implementation. Signatures using chained hex codes (CHC) were also suggested that year by Iltis et al [20]. The suggested scheme's main benefit is a size reduction, particularly in the public key and signature. The public key's compact size sped up decoding, signing, and verifying. As a possible post-quantum cryptography approach, Shrestha et al explored a polar-code-based variation of the McEliece cryptosystem in 2014 [21].

In the same year, Hooshmand suggested a public key technique using polar codes to improve the McEliece cryptosystem's efficiency further [16]. Related to the original McEliece cryptosystem, the proposed approach has an upper transmission rate  $R = 0.85$  and a smaller private and public key size  $MPB = 65.19$  bytes,  $MPR = 2.75$  kbs. Bardet et al explained the polar code's construction in 2015 which, together with a key-recovery attack provided by Shrestha [21], allows every message to be decoded. Different strategies for creating public encryption systems based on universal random linear codes were disclosed by Wang et al in the same year (2015) [8]. They demonstrated that their techniques were impervious to common assaults against linear-code-based encryption. In 2016, Moufek presented a modified version of the McEliece cryptographic system that relies on Quasi Cyclic-LDPC and Quasi Cyclic MDPC codes [24]. The generator matrix's random bits were derived from a self-shrinking generator with some tweaks. They proved that their system was safe from common structural and decoding threats. The updated McEliece cryptographic system developed by Moufek, Guenda, and Aaron Gulliver in 2017 was attacked by Dragoi et al [8].

Independent of the second code's characteristics, the attack relies solely on discovering its structure to succeed. Therefore, their conclusion holds even if a different code is used in place of the MDPC.

Developing a post-quantum secure McEliece cryptographic implementation is described in [1], which details the design process for an embedded co-processor. A co-design between hardware and software has been prioritized to ensure McEliece's success on low-cost, embedded devices in the real world. Parameters of the system, algorithms, architectural choices, and even elementary mathematical operations are all considered throughout the optimization design process. An 8-bit PicoBlaze softcore is used in the final design for its adaptability, and multiple parallel acceleration units help maximize throughput. The co-processor prototype runs on a Spartan-3an FPGA, which is only being put to 30% of its capacity. If the FPGA's clock frequency is set to 92 MHz, decrypting a single 80-bit key using McEliece takes less than 100K clock cycles or just one millisecond. The present design is ten times larger and slower than this.

The study of cryptography using codes has great promise. It paves the way for developing several cryptographic techniques, such as authentication protocols, public-key cryptosystems, etc. The McEliece cryptosystem was the first code-based public-key cryptosystem, and other variations were proposed to develop different security protocols. Radio-frequency identification systems utilize a wide variety of authentication methods, some of which are very new and rely on other implementations of the McEliece cryptosystem. These protocols are surveyed by Chikouche, Cherif, and Cayrel [2]. In addition, we analyze the safety and efficiency of each approach. Digital signature systems based on the McEliece cryptosystem are discussed in [4], as are several widely-used significant codes. Goppa codes, used in the McEliece cryptographic method, are extremely fast and were thought to be safe against multiple quantum assaults; nevertheless, the vast amount of its public keys is a serious downside. Much advancement have been made, and more are being made all the time, to lower the size of public keys without sacrificing security. This document details several enhancements and adjustments made to the McEliece cryptosystem. Pay attention to code-based cryptography's digital signature techniques as well.

It is often said that a digital computer is an effective universal computing device capable of simulating any physical computing equipment with an increase in a calculation time of, at most, a polynomial factor. In light of quantum physics, this might be different. Several suggested cryptosystems have their foundations in difficulties like factoring integers and calculating discrete logarithms, both of which have been considered challenging for a classical computer. PETER W.

SHOR examines these issues [3]. These two issues on a theoretical quantum computer are solved with efficient random methods. These methods have a runtime that scales with the input size, or in this case, the number of digits in the integer being factored. If you're looking for an alternative to the more traditional symmetric encryption method, look no further than public key cryptography, also known as an asymmetrical encryption scheme. It was in the seminal work of Hellman and Diffie that the first modern Asymmetric cryptographic scheme was introduced [5].

The Internet, a worldwide system that enables constant interaction between total strangers, was born out of their vaticinator desires. Their proposal for the DH cryptosystem was founded on a solution to a discrete logarithm problem in mathematics. The knapsack problem provides the basis for the public key cryptosystem known as Merkle-Hellman [6]. However, it is obsolete and malfunctioning. The RSA cryptosystem developed simultaneously by Rivest, Adleman, and Shamir, has yet to be cracked and is still in widespread use [6]. Multiplication of numbers, even large ones, is easy, but factorization is very difficult, which is why the RSA cryptosystem relies on it. Then, Shor created an algorithm with an exceptional trait that effectively processes integers [3]. The only catch is that it can only be used with a quantum computer. Manin and Feynman in the 1980s conceptualized the idea of a quantum computer [7] [8]. The use of quantum physics enables the quantum computer to do massively parallel processing. A quantum computer easily handles difficulties in number theory and various logarithmic problems. Research into cryptography has taken a new, post-quantum-era turn due to a string of surprising findings. One potential successor to quantum cryptography is code-based cryptography. It has a basis in linear coding theory and is connected to that field.

Golay codes first introduced in the early 1950s have recently seen a tremendous uptick in interest [25]. After 15 years of tweaks to the proposal's security parameters [26], the asymmetric encryption cryptosystem McEliece [9] presented in 1978 based on Goppa codes has remained uncracked. Niederreiter proposed the knapsack cryptosystem, based on Reed-Solomon regulations, as one such system [27]. Sidelnikov showed that the Niederreiter cryptosystem was insecure by using the Reed-Solomon and Goppa codes [28]. Sidelnikov proposed a PKC (Public Key Crypto System) using binary Reed-Muller codes [29]. It offered great safety despite its very low transmission rate (nearly 1) and simple encryption and decryption procedures. Minder cracked the Sidelnikov PKC, which relied on a previously established public key to generate a hidden one [30]. It has been revealed that the execution time of an attack that uses low-weight discovery approaches is sub exponential. Most common attacks employed to crack LDPC and QCLDPC were also examined. Londahl and

Johansson created a convolutional-code-based variant of the McEliece cryptosystem [36]. The convolutional codes used by Landais and Tillich in their assault on the McEliece cryptosystem were successful [37]. Several academics have proposed improved versions of the McEliece cryptosystem that use error-correcting codes other than Goppa codes, such as AGC, LDPC, and convolutional codes. Despite this, it has been shown that each approach is insecure, leading to the widespread use of Goppa codes.

## 2.1 Limitation of the existing literature

Table 1 shows some weaknesses/ limitations of the existing literature:

Table 1: Limitations of Existing Literature

<u>S. No</u>	<u>Limitations</u>	<u>Detail</u>
1	In the existing literature, there are rare evidences where standard McEliece encryption has been analyzed/ improved just by changing the parameters instead of changing coding family.	There is an exhaustive list where the researchers proposed different coding schemes /families other than Goppa codes to improve the McEliece cryptosystem by decreasing public and private key lengths. However, there are rare evidences where problem of original McEliece cryptosystem based on Goppa codes is addressed by varying parameters.
2	Existing literature is lacking in implementation of McEliece encryption in any practical application.	Existing literature is lacking in implementing original McEliece cryptosystem based on Goppa codes in any telecommunication standard/application. However, the proposed use case of McEliece cryptosystem in this research paper is quantum resistive and can be used as an alternate of chat app .

## **Post Quantum Cryptography**

### **3.1 Introduction**

Quantum-proof, quantum-safe, and quantum-resistant are all terms that may be used to describe the encryption methods used in the post-quantum era. Cryptographic approaches (often public-key algorithms) that are assumed to be secure against the cryptanalytic assault of a quantum computer are the focus of this field of study. Popular methods rely on solutions to the integer factorization, discrete logarithm, and elliptic curve discrete logarithm issues, all of which have a fundamental flaw. A sufficiently powerful quantum computer may use Shor's approach to solve these issues in a matter of seconds [55] [57].

Although quantum computers lack the processing ability to break any practical cryptographic technique at the present time,[58] several cryptographers are working on alternative solutions just in case. Since 2006, academics and crypto business experts have gathered at PQCrypto conferences. The European Telecommunications Standards Institute (ETSI) and the Institute for Quantum Computing have recently given talks on Quantum-Safe Cryptography. While quantum computers pose a significant risk to traditional public-key protocols, most contemporary symmetric cryptographic algorithms and hash functions are thought to be mostly immune to such assaults[59][61] [57] [62]. Even if attacks on symmetric cyphers are sped up using the quantum version of Grover's technique, increasing the key size by two may effectively circumvent this [63]. As a result, despite the arrival of quantum computers, symmetric cryptography implementations will not need significant changes. However, existing asymmetric/ public key crypto systems like RSA and ECC will no longer be secure once quantum computers become practical. So, there must be alternate public key cryptosystems/ algorithms that are equally good and secure post quantum world as well.

## 3.2 Explanatory Chart

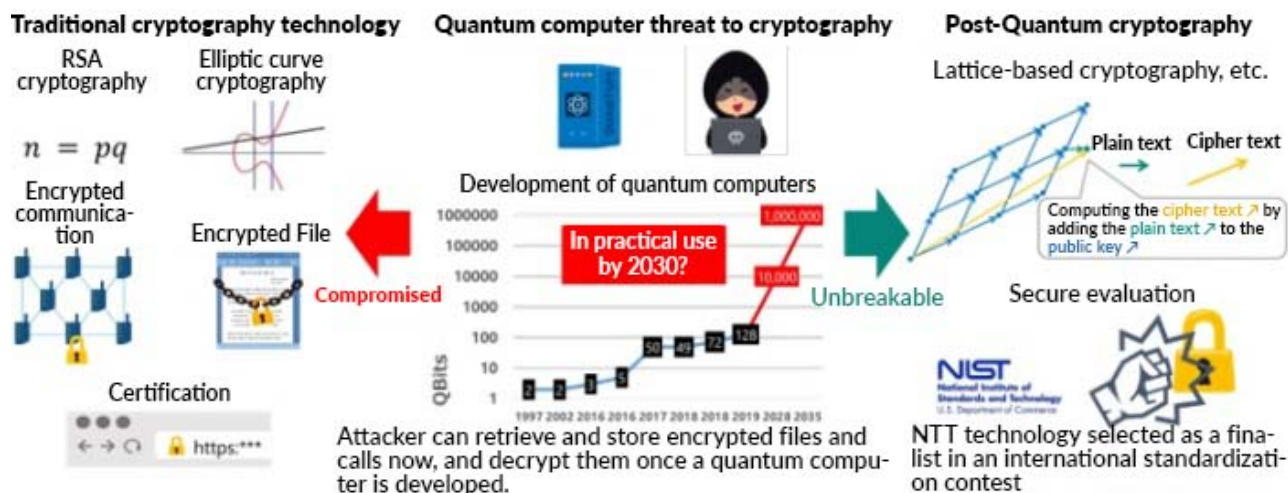


Figure 1: Explanatory Chart [8]

## 3.3 Algorithms

There are now six main areas of concentration in post-quantum cryptography research.

### 3.3.1 Lattice-based cryptography

Several different cryptographic methods use this method, from the more traditional NTRU or GGH encryption to the more modern NTRU signature and BLISS signature, and everything in between [67]. The NTRU encryption system, for example, has been researched for a long time, but no one has yet found a practical attack on it. The security of other algorithms, such as the ring-LWE methods, is a worst-case concern [68] [69] [70]. NTRU was still under patent protection at the time. Some research suggests that NTRU's security features are even more robust than other lattice-based algorithms [71].

### 3.3.2 Multivariate cryptography

This category covers cryptographic schemes that rely on the complexity of terminal access controller access control of multivariate equations, like the Rainbow (Unbalanced Oil and Vinegar) scheme. Many insecure encryption techniques have been developed for multivariate equations. It is possible, however, that multivariate signature methods like Rainbow form the foundation of a quantum-safe digital signature [72]. A patent protects Rainbow signatures.

### **3.3.3 Hash-based cryptography**

This class of cryptographic systems includes, but is not limited to, Lamport signatures, Merkle signature schemes, XMSS,[73] SPHINCS[74], and WOTS schemes. However, there has been a rebirth in the use of such signatures in recent years due to growing interest in quantum-resistant encryption. Merkle signatures and any hash techniques used in conjunction with them are not patentable. The XMSS stateful hash-based signature mechanism created by Johannes Buchmann's team is described in RFC 8391[75]. In the past, systems were either one-time-use or restricted in their potential applications. UOWHF hashing was devised in 1989 by Moni Naor and Moti Yung, who also developed a hash-based signature (the Naor-Yung scheme) that may be used indefinitely[76].

### **3.3.4 Code-based cryptography**

Error-correcting codes are used by many encrypted systems, like the McEliece and Niederreiter encryption algorithms and the Courtois, Finiasz, and Sendrier Signature methods that go with them, to make sure that their messages are secure. The original McEliece signature, which was sealed with random Goppa codes about 40 years ago, has not been decrypted yet. McEliece has been shown to be unsafe, even though the code has been improved to make it more organized and the size of the keys has been cut down. The Post-Quantum Cryptography Study Group, which is funded by the European Commission, has found that the McEliece public key encryption method could be used to protect against future attacks that use quantum computers[69].

### **3.3.5 Super singular elliptic curve isogeny cryptography**

This encryption scheme is a forward-secrecy alternative to Diffie-Hellman that makes use of supersingular elliptic and super singular isogeny graphs. This cryptographic implementation of a key exchange mechanism similar to Diffie-Hellman is based on the well-studied mathematics of super singular elliptic curves. Since the widely-used Diffie-Hellman and elliptic curve Diffie-Hellman key exchange techniques are vulnerable to quantum computing, this is a straightforward substitute[78]. It functions similarly to existing Diffie-Hellman implementations; therefore, it provides forward secrecy, which is crucial for preventing both widespread government surveillance and the accidental disclosure of long-term keys [79]. In 2012, scientists Sun, Tian, and Wang from Xidian University and the Chinese State Key Lab for Integrated Service Networks created digital signatures that were immune to quantum attacks. De Feo, Jao, and Plut served as inspiration for their art [80]. No patents exist to protect this method of encryption.

### 3.3.6 Symmetric key quantum resistance

Symmetric-key algorithms, such as AES and SNOW 3G, are unbreakable by a quantum computer with sufficiently high key sizes[81]. A quantum computer might theoretically break public key encryption. Symmetric key cryptography, on the other hand, is safe enough to be used by Kerberos and the 3GPP Mobile Network Authentication Structure, two popular key management tools and systems. Post-quantum cryptography may be implemented quickly and easily, according to some studies, by expanding the usage of symmetric key management systems like Kerberos [82].

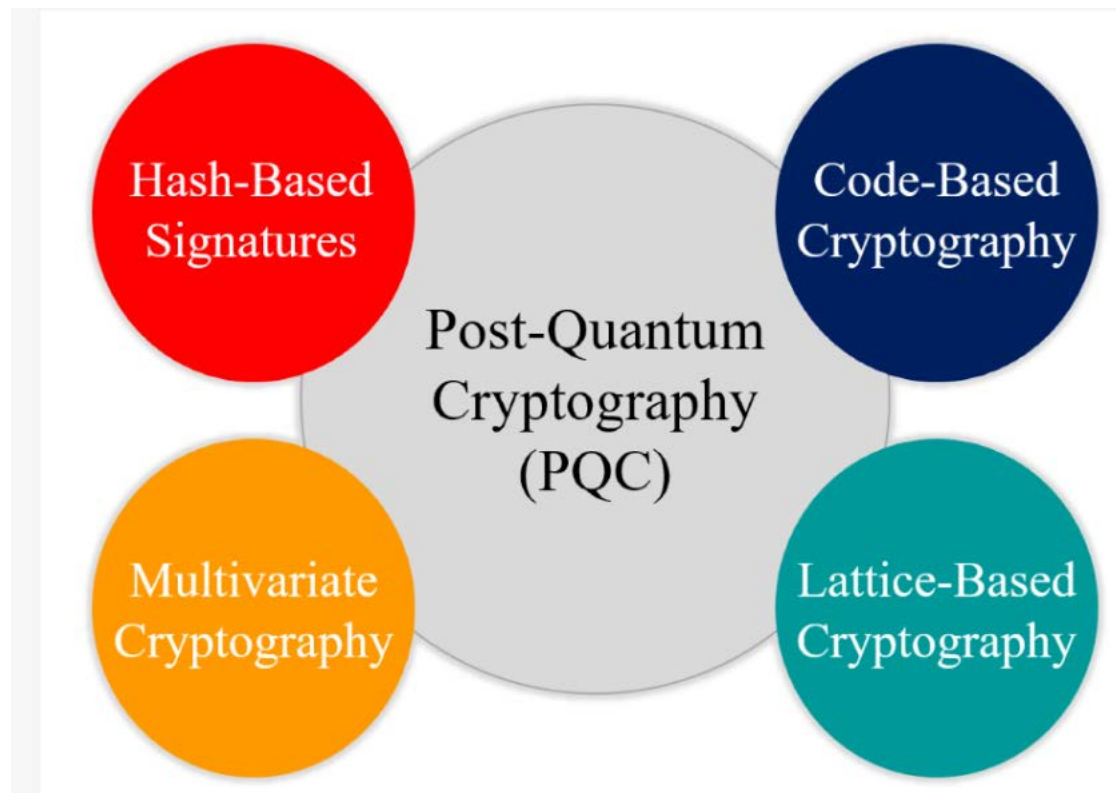


Figure 2: Types of Post Quantum Cryptography [69]



### 3.4 Key Attributes of PQC Algorithms

Few of Key attributes of above-mentioned schemes/ algorithms are mentioned in Table 2: -

Table 2: Key attributes of PQC Schemes/algorithms

<u>PQC Algorithm Family</u>	<u>Function/Use</u>	<u>Example</u>	<u>Notable Attributes</u>
Hash-based Cryptography	Digital signatures	Digital signatures	Well-understood. Stateful schemes needed to reduce large signature sizes.
Lattice-based Cryptography	KEM/Encryption, Digital signatures	FrodoKEM, NewHope, NTRU, FALCON, qTESLA	Short ciphertext and keys, good performance, sometimes complex. Short signatures.
Code-based Cryptography	KEM/Encryption	BIKE, Classic McEliece, HQC, NTS-KEM, RQC	High confidence. Fast encryption but larger public keys.
Multivariate Cryptography	Digital signatures	EMSS, LUOV, MQDSS, Rainbow	Large key sizes (~1 MB / ~11 KB). Schemes need more analysis
Supersingular Elliptic Curve Isogeny Cryptography	KEM/Encryption	SIKE	Very small key sizes (less than 500 B), slower performance, relatively new.

### 3.5 Performance Comparison

Table 3: Quantum Resistive algorithms comparison

Algorithm	Advantages	Disadvantages
Lattice based Cryptography [91]	<ul style="list-style-type: none"> <li>• Fast and Efficient due to linear algebraic based matrix and vector operations on integers</li> <li>• Small key size</li> <li>• Can be used in other security services like homomorphism encryption, identity-based encryption and attribute-based encryption</li> <li>• More versatile and can</li> </ul>	<ul style="list-style-type: none"> <li>• Computationally intensive so not suitable for resource constraint devices</li> <li>• Difficult to give accurate estimations of the security level on software and hardware as its comparatively new and active area of cryptography and future advancement can weaken the security of lattice-</li> </ul>

	<p>be used in encryption, digital signature and key exchange etc</p>	<p>based schemes.</p> <ul style="list-style-type: none"> <li>• Hardware implementation is vulnerable to physical attacks such as timing attacks, fault attacks and power analysis</li> </ul>
<p><b>Multivariate cryptography</b> [91]</p>	<ul style="list-style-type: none"> <li>• Fast and good approach for signatures, it offers shorter signatures than those offered by other schemes</li> <li>• Potential for efficient implementations</li> <li>• Relatively small key sizes</li> </ul>	<ul style="list-style-type: none"> <li>• Vulnerability to advancements in algebraic techniques</li> <li>• Security is not assured as quite a number of multivariate cryptosystems have been broken over the years</li> <li>• Limited number of practical schemes</li> </ul>
<p><b>Hash-based cryptography</b> [91]</p>	<ul style="list-style-type: none"> <li>• Fast as they need only computer hash functions</li> <li>• Quite secure as hash function is resistant to collision and primage attacks</li> <li>• Efficient implementations possible</li> </ul>	<ul style="list-style-type: none"> <li>• Record of previous signature is required, if not done it creates insecurity in large environments</li> <li>• Can produce a limited number of signatures, if number is increased the signature size increase that is ineffective</li> <li>• Limited versatility</li> </ul>

		compared to some other schemes
<b>Code Based Cryptography [91]</b>	<ul style="list-style-type: none"> <li>• It has proven security as underlying mathematical problem related to error correcting codes has long history and have been extensively studied.</li> <li>• Fast in doing encryption and decryption. So computationally more efficient than other schemes.</li> <li>•</li> </ul>	<ul style="list-style-type: none"> <li>• Large key length difficult to implement with low processing power and less storage devices</li> </ul>
<b>Supersingular elliptic curve isogeny cryptography</b>	<ul style="list-style-type: none"> <li>• Efficient key exchange protocols</li> <li>• Relatively small key sizes</li> </ul>	<ul style="list-style-type: none"> <li>• Complex mathematics and implementation requirements</li> <li>• Slower performance</li> <li>• Vulnerability to potential cryptanalytic advances</li> <li>• Relatively new</li> </ul>

### **3.6 Security reductions**

Proving the equivalency of a cryptographic technique to a well-known difficult mathematical equation is the desired goal in the field of cryptography research. Security reductions are proofs used to show how tough it is to break an encryption method. A cryptographic algorithm's security may be limited to protecting some well-known hard problem. For the sake of more secure post-quantum cryptography, researchers are aggressively searching for potential security holes. The most recent findings are shown below:

#### **3.6.1 Lattice-based cryptography – Ring-LWE Signature**

Some ways to use Ring-LWE make it easier to use by reducing security to solving the shortest-vector problem (SVP) in a lattice. Everyone knows that the SVP is NP-hard[83]. It has been shown that some ring-LWE systems, like the one described by Güneysu, Lyubashevsky, and Poppelmann, reduce security in a way that can be proven[65]. The GLP signature made by Güneysu, Lyubashevsky, and Poppelmann was changed into the GLYPH signature so that it could include new information learned from further study. The GLP signature was made public for the first time in 2012. The Ring-TESLA signature can be used instead of the Ring-LWE signature[84]. Learning with Rounding (LWR) is a "DE randomized variation" of LWE that makes efficiency and bandwidth better "by avoiding tiny sampling mistakes from a Gaussian-like distributed with deterministic errors" [85]. LWE rounds up to hide the smaller numbers, while LWR rounds down to show them.

#### **3.6.2 Lattice-based cryptography – NTRU, BLISS**

The Closest Vector Problem (CVP) in a Lattice is believed to be related to the security of the NTRU encryption technique and the BLISSsignature[67]. However, this issue cannot be reduced to mere evidence. The NP-hardness of the CVP has long been accepted as fact. For long-term use, the Post Quantum Cryptography Study Group recommended the less secure Stehle-Steinfeld variant of NTRU. This research was financed by the European Commission [69].

#### **3.6.3 Multivariate cryptography – Unbalanced Oil and Vinegar**

Unbalanced Oil and Vinegar signature methods are the asymmetric cryptographic counterpart of cavemen. They are based on multivariate polynomials over a finite field show style.  $\mathbb{B}$   $\mathbb{F}$ . Bulygin, Petzoldt, and Buchmann showed that solving the NP-Hard Multivariate Quadratic Equation Solving problem is the same as solving general multivariate quadratic UOV systems[86].

### **3.6.4 Hash-based cryptography – Merkle signature scheme**

In 2005, Luis Garcia demonstrated that the integrity of a Merkle Hash Tree signature is only as good as the underlying hash algorithm. Garcia demonstrated in his article that one-way hash functions may be used to guarantee the security of a Merkle Hash Tree signature [87].

The security of the Merkle tree signature may be compromised by the use of a hash function with such a reduction to a known issue, however this is something that can be verified.[88] Merkle signatures have been proposed as a long-term security mechanism against quantum computers by the European Commission-funded Post Quantum Cryptography Study Group [69].

### **3.6.5 Code-based cryptography – McEliece**

The Syndrome Decoding Problem (SDP) exposes a weakness in the McEliece encryption mechanism. The SDP's NP-hardness has been extensively documented[89]. The Post Quantum encryption Study Group, funded by the European Commission, has argued in favor of adopting this encryption to protect against the eventual development of quantum computers[69].

### **3.6.6 Code-based cryptography – RLCE**

Using the same principles as the McEliece techniques, Wang presented the RLCE random linear code encryption scheme in 2016 [90]. Any linear code, including the Reed-Solomon code, may create an RLCE method by introducing random columns into the underpinning linear code generator matrix.

### **3.6.7 Super singular elliptic curve isogeny cryptography**

The construction of an isogeny among two super singular curves of equal point count is relevant to the issue of security. Recent research on the problem's complexity by Delfs and Galbraith confirms that it is as challenging as the creators of the encrypting and decrypting claim it to be. Security cannot be reduced to a known NP-hard issue [91].

## **3.7 Comparison**

Many algorithms developed after the advent of quantum mechanics call for greater key sizes than the conventional "pre-quantum" public key methods. Key size, computational efficiency, and the size of the cipher text or signature are all factors that must frequently be balanced. There are several values for various schemes in the table 3, all of which have a post-quantum security level of 128 bits.

**Table 3: Quantum Resistive Algorithms Comparison**

<u>Algorithm</u>	<u>Type</u>	<u>Public Key</u>	<u>Private Key</u>	<u>Signature</u>
<b>NTRU Encrypt</b>	<b>Lattice</b>	<b>766.25 B</b>	<b>842.875 B</b>	
<b>Streamlined NTRU Prime</b>	<b>Lattice</b>	<b>154 B</b>		
<b>Rainbow</b>	<b>Multivariate</b>	<b>124 KB</b>	<b>95 KB</b>	
<b>SPHINCS</b>	<b>Hash Signature</b>	<b>1 KB</b>	<b>1 KB</b>	<b>41 KB</b>
<b>SPHINCS</b>	<b>Hash Signature</b>	<b>32 B</b>	<b>64 B</b>	<b>8 KB</b>
<b>BLISS-II</b>	<b>Lattice</b>	<b>7 KB</b>	<b>2 KB</b>	<b>5 KB</b>
<b>GLP-Variant GLYPH Signature</b>	<b>Ring-LWE</b>	<b>2 KB</b>	<b>0.4 KB</b>	<b>1.8 KB</b>
<b>NewHope</b>	<b>Ring-LWE</b>	<b>2 KB</b>	<b>2 KB</b>	
<b>Goppa-based McEliece</b>	<b>Code-based</b>	<b>1 MB</b>	<b>11.5 KB</b>	
<b>Random Linear Code based encryption</b>	<b>RLCE</b>	<b>115 KB</b>	<b>3 KB</b>	
<b>Quasi-cyclic MDPC-based McEliece</b>	<b>Code-based</b>	<b>1,232 B</b>	<b>2,464 B</b>	
<b>SIDH</b>	<b>Isogeny</b>	<b>564 B</b>	<b>48 B</b>	
<b>SIDH (compressed keys)</b>	<b>Isogeny</b>	<b>330 B</b>	<b>48 B</b>	
<b>3072-bit Discrete Log</b>	<b>not PQC</b>	<b>384 B</b>	<b>32 B</b>	<b>96 B</b>
<b>256-bit Elliptic Curve</b>	<b>not PQC</b>	<b>32 B</b>	<b>32 B</b>	<b>65 B</b>

The time and energy needed to transmit public keys over the internet is a real-world factor that should be considered while deciding between post-quantum cryptography methods.

## McEliece cryptosystem

### 4.1 Introduction

Communication systems that are both reliable and safe have been in use since far before the Roman Empire's zenith. A cryptosystem is a system for sending information in such a way that only the recipient can decipher it. Interception during transmission, mistakes, and whether or not a technique is even usable are just a few examples of the many challenges encountered with these approaches. Goppa codes, the first codes employed in the McEliece cryptosystem, are an example of error-correcting codes. In this chapter, we'll look at the safety of the original McEliece cryptosystem, which makes use of Goppa codes, and explain how they work.

Messages don't always get through as expected. The prevalence of mistakes is largely to blame for this. Multiple factors, including chance, neighboring channels, and external interference, may contribute to transmission errors. Message faults may be detected and fixed via error-correcting codes. Including redundancy in the message ensures that the intended meaning of the message is sent even if a mistake occurs in its transmission. For whatever reason, the receiver hears "I'm bringing my brushes" instead of "I'm bringing my gloves." An erroneous communication may be received without the intended meaning being understood by the recipient. Now imagine if there were some more redundancy in the message, like "Because it is snowing severely outside, I am going to carry my gloves." This bolstered safety net allowed the receiver to properly infer that the sender intended to indicate he or she would bring something to keep warm, and that the message received was a mistake[4].

The Goppa code is an error-correcting coding scheme that uses modular arithmetic, in which an increasing series of integers is repeated until the desired value is reached, at which point the process begins again from zero. One useful use of modular arithmetic is the use of a 24-hour clock rather than a 12-hour clock for maintaining time. Modulo 24 is used to represent the hour on a 24-hour clock, whereas modulo 12 is used on a 12-hour clock. It doesn't matter whether you use a 12-hour or 24-hour clock, the time at three in the morning is the same either way. Even though 15:00 on a 24-hour clock corresponds to 3:00 on a 12-hour clock, the latter "wraps back around," making each hour above 12 a multiple of 3[5].

A key generation algorithm, an encryption technique, and a decryption technique are the basic

minimum for every cryptosystem. Suppose, for the purpose of argument that Alice and Bob want to hold a private conversation but are aware that a third person, Eve, may be listening in. Alice wants to get in contact with Bob. Alice encodes the plaintext of her message to Bob using a key. Bob may be able to decipher the original message from the cipher text if he finds the key. We always simply assume that Eve is familiar with the overall procedure but not the essential details. While Eve instructs us to safely infer that she wants us to study the message, locate the key and decode all communications encrypted with that key, alter Alice's message, or pretend to be Alice while corresponding with Bob.

The McEliece cryptosystem requires two different keys—public and a private—to encrypt and decrypt a message. Let's say Bob freely shares his public key online for anyone to see. With Bob's public key, Alice may send him an encrypted message. Bob has to use his private key in order to decrypt the communication. In order for this to be useful, however, it requires that Eve needs more than simply a public key in order to decrypt messages. Instead, Bob would need to provide Eve the secret key to decipher the cipher text[18].

Bob produce a public key for use in the McEliece Cryptosystem by selecting a Goppa polynomial  $(z)$  of degree  $t$  and calculating the generator matrix  $G$  of the Goppa code. "After that, Bob would choose two invertible and permutable matrices,  $S$  and  $P$ , at random. and plug them into the equation  $G' = SGP$ . Bob's public key would be  $(G', t)$ , while his private key is  $(S, G, P)$ .

Alice's first step would be to construct her message as a series of binary strings that would be sent out as the final encrypted message, which contains a random error vector with a weight of  $t$  or less  $y = mG' + e$  added to it.

Next, Bob would utilize his  $P$  matrix to get  $y' = yP$ . Then, Bob would apply the Goppa code  $G$  decoding technique to turn  $y'$  into the right code word  $m' = mS$ .with the help of  $S$ , Bob would derive the original message as  $m = m'S$ .

It is very unlikely that Eve would be able to understand Alice's message without her secret key. This is because she needs to separate matrix  $G$  from its mirror image, matrix  $G'$ . Since Eve can't just look up the inverse of the matrix  $G'$ , she'd need to know the inverse of the unpublished random matrix  $S$  as well. Eve doesn't know what the matrix  $P$ , thus she has no notion where to seek for  $y'$  to get  $m'$ . The security of this cryptosystem depends on how difficult it is to interpret  $y'$  and find  $m'$ . This task makes use of a massive Goppa code. For instance, in his first cryptosystem work published in 1978, McEliece suggested a  $[1024, 524]$  Goppa code (i.e., a



Goppa code of length 1024 and dimension 524).

The bigger the code, however, the less useful the cryptosystem becomes; therefore, this is a serious problem. This cryptosystem is currently not very practical, however that should change as technology and storage space improve [33].

The system is also more susceptible to attack since the same encryption matrix  $G'$  will be used to deliver the same message several times. Unlike the Rivest-Shamir-Adleman cryptosystem, this one doesn't come with a clear method of creating signatures (RSA). Since anybody may use Bob's public key to send him a message and has no way of knowing whether the sender was indeed Alice unless they both know a shared Goppa Code.

The system's benefits, on the other hand, include the fact that it is one of the simplest cryptosystems out there and has been extensively researched ever since it was first introduced in 1978. The use of this technology also enables quick times for both encrypting and decrypting data.

## **4.2 Part I: Basic Terminology**

### **4.1.1 Cryptology**

The study of cryptology centers on the transmission and reception of information securely, without the possibility of interception, reading, or modification by a third party [25].

Think about Alice and Bob, two pals of ours. They want to talk to one other secretly, but they're worried that a person called Eve could listen in. Say Alice has to contact Bob and she decides to use a messenger service. Alice transforms the plaintext of her communication into the cypher text, a sequence of code words, using a cryptosystem that has been mutually agreed upon by the two friends. It is often assumed that Eve is aware of the particular cypher or cryptosystem being used, and that the only thing preventing her from reading the message is the key or keys being used to encrypt and decrypt it.

It's important to note that the process of encoding and decoding a message may be variable based on the cryptosystem in use. The class of cryptosystems known as Public Key Cryptosystems is one such example [73].

Each of these forms of cryptography relies on a pair of keys—a public one and a private one. Bob shares his public key online, where it may be seen by anybody, including Alice and Eve. To transmit the message to Bob, Alice uses the public key to encrypt it, and Bob receives it and

decodes it using his private key. However, Eve cannot read the message even if she has the public key because of the mechanism employed. Private keys are required to decode cypher texts, and Bob is the only person who has access to the private key since he is the intended recipient of the cypher text[69].

#### 4.1.2 Error Correcting Codes

It's unclear what Alice was trying to tell Bob in her message. So, what happened? Obviously, Eve was making a clumsy effort at diversion here, or the letter got corrupted in transit. The purpose of error correcting codes (ECC) is to identify and fix transmission errors. In order to do this, redundancy is added to the message to ensure its clarity in the event of a misinterpretation. As soon as a message is encoded, it is transformed into a code word that includes both the original message and the redundancy. Certain error rates that can be tolerated by the decoding algorithms are used for these codes. This decoding technique allows for the correction of transmission mistakes and the subsequent restoration of the original message. This gives the cipher text  $y = c + e$ , where  $c$  is the code word and  $e$  is an error vector. This is helpful because Eve would have had a hard time figuring out the secret message from the coded text she got.

#### 4.1.3 Fields

An essential component of mathematics, a field enables us to classify numbers into distinct categories.[10]

**Definition 1.3.1** A field  $F$  is a collection of elements that are closed under two processes and meet the following conditions:

1. *There exists an element  $a \in F$  such that, for all  $x \in F$ ,  $x + a = x$ .*
2. *There exists an element  $b \in F$  such that, for all  $x \in F$ ,  $xb = x$ .*

*For all  $x, y, z \in Z$ ,*

3.  $x + y = y + x$
4.  $xy = yx$
5.  $(x + y) + z = x + (y + z)$
6.  $(xy) = (yz)$
7.  $(y + z) = xy + xz$
8. *For each  $x$ , there exists an element  $-x$  such that  $x + (-x) = a$ .*
9. *For each  $x \neq a$ , there exists an element  $x^{-1}$  such that  $xx^{-1} = b$ .*

**Example 1.3.1.** The numbers, which are written as  $Z \dots 3, 2, 1, 0, 1, 2, 3, \dots$ , are not an area under

(+). This is because, besides the numbers 1, 1, and 0, none of the other integers meet property number 9.

**Example 1.3.2** The set of rational numbers

$$Q = \{a/b \text{ such that } a, b \in Z \}$$

is a field closed under (+, ·) and as such, upholds the previous nine rules.

1.  $y = \frac{0}{z}$  for any element  $z \neq 0$  so that  $\frac{0}{z} + x = x$
2.  $y = \frac{1}{1}$  such that  $\frac{1}{1} \cdot x = x$

For all  $a, b, c, d \in Z$

3.  $\frac{a}{b} + \frac{c}{d} = \frac{c}{d} + \frac{a}{b}$
4.  $\frac{a}{b} \cdot \frac{c}{d} = \frac{c}{d} \cdot \frac{a}{b}$
5.  $\left(\frac{a}{b} + \frac{c}{d}\right) + \frac{e}{f} = \frac{a}{b} + \left(\frac{c}{d} + \frac{e}{f}\right)$
6.  $\left(\frac{a}{b} \cdot \frac{c}{d}\right) \cdot \frac{e}{f} = \frac{a}{b} \cdot \left(\frac{c}{d} \cdot \frac{e}{f}\right)$
7.  $\frac{a}{b} \cdot \left(\frac{c}{d} + \frac{e}{f}\right) = \frac{a}{b} \cdot \frac{c}{d} + \frac{a}{b} \cdot \frac{e}{f}$
8. For each  $\frac{a}{b} \in Q$ , one has  $\frac{a}{b}^{-1} + \frac{a}{b} = 0$
9. For each  $\frac{a}{b} \in Q$ , with  $a \neq 0$ , the element  $\frac{a}{b}$  satisfies  $\frac{a}{b} \cdot \frac{b}{a} = 1$

When the number of possible values in a field is finite, we say that it is a Galois field. A Galois field is denoted by the notation GF(q), where q is the field's order [12].

**Theorem:** Assume p is prime. There is only one finite field of order pm, and this holds true for all powers pm.

Suppose p is prime, k is positive and less than Z, and  $p > 0$ . The Galois field of order  $q = pk$ , where p and k are the numbers of elements in the field, is denoted by the notation GF(pk), which is the extension Galois field of (p) of degree m[8].

An irreducible polynomial over (pm) is one that cannot be factored into a lower degree polynomial over (pm) [6].

**Example:** Multiply  $1 + 3$  by (2). X and/or X<sup>2</sup> must be included in any polynomial with a degree lower than 3. There is no division by such a polynomial. the answer is  $1 + X + X^3$ , etc. An irreducible polynomial over (2) is given by  $(1 + X + X^3)$ .

**Example:**  $X + X^5$  is not an irreducible polynomial over  $(2)$  because

$$\frac{X + X^5}{X} = 1 + X^4$$

Let's pretend  $x$  is an irreducible  $m$ -degree polynomial over  $GF(p)$ . An example of a primitive polynomial if  $n = p^m - 1$  is the smallest possible integer for which  $(x)$  divides  $X^n - 1$  [3].

**Example:** Suppose  $(x) = 1 + X + X^3$  with degree 3 over  $GF(2)$ .  $n = 2^3 - 1 = 7$  and so we have  $X^7 - 1$ .  $X^7 - 1$  can be factored into irreducible polynomials as  $(X + 1)(1 + X + X^3)(X^3 + X^2 + 1)$ . It can be checked that  $(x)$  does not divide  $X^v$  such that  $v < 7$ .

Therefore,  $1 + X + X^3$  is a primitive polynomial of degree 3 over  $(2)$ .

**Definition:** Integers  $a$  and  $b$  are said to be congruent modulo  $n$ , written  $a \equiv b \pmod{n}$ , if and only if  $a - b = kn$  for any integer  $k$ , where  $k$  is an integer greater than or equal to  $n$ .

**Example:** Create a congruent representation of 49 modulo 5, where  $49 - b$  is evenly divisible by 5. Finding a value of  $k$  such that is required by the definition.  $49 - b = 5k$ . Finding 5,000, though, is sufficient. 45, 40, and 35 are all multiples of 5, hence  $b = 4, 9,$  and  $14$  accordingly.

We want to find a value for  $b$  such that.  $0 \leq b < 5$ . The term "smallest possible non-negative residue" is used to describe this  $b$ . When  $b = 4$ , the residue is the minimum it can be without being negative. This would be represented as  $49 \equiv 4 \pmod{5}$ .

**Example:** Write  $81 \equiv b \pmod{2}$  such that  $b$  is the smallest possible non-negative residue.

$$81 - b = 2k$$

$$81 - b = 2 \quad (40)$$

$$b = 1$$

Therefore,  $81 \equiv 1 \pmod{2}$ .

**Example:**  $(2) = \mathbb{Z}_2 = \{0,1\}$  = The integers modulo 2. This is also known as the binary field.

#### 4.1.4 Binary

As a common practice we reduce data to a series of 1s and 0s before feeding it into a central processing unit. Binary encoding is used to describe this shift. Our usual practice is to think about digits in the 10s-base system. Despite the fact that binary uses a base 2 representation for numbers. Each component of a power of 2 may be either 0 or 1, every integer can be written as an additive sequence of power of 2 when converted into binary number. Putting the coefficients in descending order yields a binary representation of the number [67].

In binary, only 0 and 1 are utilized; they have the additive qualities of components modulo 2 ( $0+1=1$ ,  $1+1=0$ ,  $0+0=0$ , and  $1+1=0$ ).

#### 4.1.5 Hamming Distance

**Definition:** A code word, lawful code word, or code vector is a subset of code words represented by the notation  $= (c_1, c_2, \dots, c_n)$ . If a sequence of length  $n$  is in  $A^n$  but not in a code, it is considered an unlawful code word.

The binary digits 0 and 1 will be assumed to make up alphabet  $A$ . The playing field is the alphabet  $A = \{0, 1\}$ . Modulo 2: the integers. A code constructed using this alphabet is called a binary code [87].

If we make a mistake, we can fix it more easily if the code words are spread widely apart from one another. Otherwise, an accidental transposition of letters might change the meaning of the code. To prevent this, the forbidden code phrases are usually placed at random intervals among the permitted ones. In doing so, we increase the Hamming distance between the two words.

If message  $m$  contains the code words  $c_i$  and  $c_j$ , then the distance between these two words is the Hamming distance. It is quantified by tallying the number of bits that are distinct between the two code phrases [65].

$C$  is the smallest Hamming distance possible between two lawful code words,  $c_i$  and  $c_j$ . The smallest Hamming distance from  $C$  is denoted by the symbol  $C$ . This demonstrates how well a code handles errors [25].

For a code word  $c_i$ , the *Hamming weight*, denoted  $(c_i)$ , is the number of nonzero places in  $c_i$ . This is the amount of ones in a binary coding word [48].

### 4.1.6 Linear Codes

**Definition:** A linear code over a field of  $k$  dimensions and  $n$  length is a  $k$ -dimensional vector space subset.  $C$  is a  $[n, k]$  code for an array of  $n$ -dimensional vectors. We call a code an  $[n, k, d]$  if its minimum Hamming distance is at most  $d$ . To this goal, we'll refer to a sequence of  $2k$  binary digits (a binary code of length  $n$  and dimension  $k$ ) as a linear code [4].

### 4.3 Goppa Codes

In order to encrypt and decode messages, we utilize a Goppa code, which is a linear, error-correcting code. This definition applies to any such code: [3]

**Definition:** Polynomials over  $(p^m)$  are what we'll call "Goppa polynomials," and we'll define them as such,

$$g(x) = g_0 + g_1x + \dots + g_t x^t = \sum_{i=0}^t g_i x^i$$

with each  $g_i \in GF(p^m)$ . Let  $L$  be a finite subset of the extension field  $(p^m)$ ,  $p$  being a prime number, say

$$L = \{ \alpha_1, \dots, \alpha_n \} \subseteq GF(p^m)$$

such that  $g(\alpha_i) \neq 0$  for all  $\alpha_i \in L$ . Given a code word vector  $c = (c_1, \dots, c_n)$  over  $GF(q)$ , we have the function

$$R_c(z) = \sum_{i=1}^n \frac{c_i}{x - \alpha_i}$$

Where

$$\frac{1}{x - \alpha_i}$$

Is a unique polynomial with  $(x - a_i) * \frac{1}{x - a_i} \equiv 1 \pmod{g(x)}$  degree less than or equal to  $t$  minus 1. After that, a Goppa code. Each code vector  $c$  in  $(L, g(x))$ . Satisfies the condition that  $(x) \equiv 0 \pmod{g(x)}$ . If  $x$  divides a polynomial, then the polynomial is divisible by  $g(x)$ .

#### 4.2.1 Parameters

**Theorem:** Goppa code's  $k$ -dimension.  $(L, g(x))$  with length  $n$  is bigger than or equal to  $n - mt$ , or  $k \geq n - mt$ .

**Theorem:** Minimal conceivable Goppa distance  $(d)$ . The  $n$ -length sequence  $(L, g(x))$  is greater than or equal to  $t + 1$ , that is  $d \geq t + 1$ .

#### 4.2.2 Binary Goppa Codes

A binary Goppa code is supposed to be written as  $\Gamma(L, g(x))$  have to use a polynomial  $g(x)$  over  $(2^m)$  of degree  $t$  [28].

**Theorem 2.3.1** Any irreducible, binary Goppa code  $\Gamma(L, g(x))$  has a least distance  $d$  of greater than or equal to  $2t + 1$ , that is,  $d \geq 2t + 1$ .

The parameters are supposed to be as written below:  $[n, k \geq n - mt, d \geq 2t + 1]$ .

#### 4.2.3 Parity Check Matrix

Parity check matrixes are used in Goppa codes to decode and recover the original message as sent by the receiver.

**Proposition:** If we set  $H = XYZ$  such that

$X = \begin{pmatrix} g_t & 0 & 0 & 0 & g_{t-1} & g_t & 0 & 0 & \cdots & g_1 & g_2 & g_3 & g_t \end{pmatrix}$ ,  $Y =$   
 $\begin{pmatrix} 1 & 1 & 1 & a_1 & a_2 & a_n & \vdots & \vdots & a_1^{t-1} & a_2^{t-1} & a_n^{t-1} \end{pmatrix}$ , and  $Z =$   
 $\begin{pmatrix} \frac{1}{g(a_1)} & 0 & 0 & 0 & \frac{1}{g(a_2)} & 0 & \cdots & 0 & 0 & \frac{1}{g(a_n)} \end{pmatrix}$ , then matrix H is a parity check matrix for a Goppa Code  $\Gamma(L, (x))$

Since  $(x)$  is irreducible, there exists a primitive element  $\alpha$  for all  $\alpha \in (2^m)$  in such a way that  $g(\alpha) \neq 0$ . Therefore subset  $L$  can encompass all basic elements of  $(2^m)$ .

$$\frac{g(x) - g(a_i)}{x - a_i} = \sum_{j=0}^t a_j \cdot \frac{x^j - a_i^j}{x - a_i} = \sum_{w=0}^{t-1} a_w X^w \sum_{j=w+1}^t a_j a_i^{j-1-w}, \text{ for all } 1 \leq i \leq n+1$$

There is supposed to be an arbitrary vector  $c \in \Gamma(L, g(x))$  if and only if

$$\sum_{i=1}^n a_i \left( \frac{1}{g(a_i)} \sum_{w+1}^t a_w g_j a_i^{j-1-w} \right) \cdot c_i = 0 \text{ for all } w = 0, \dots, t-1$$

So we can write the parity check matrix denoted as H ( $H = XYZ$ ), where X =

$$X = \begin{pmatrix} g_t & 0 & 0 & 0 & g_{t-1} & g_t & 0 & 0 & \cdots & g_1 & g_2 & g_3 & g_t \end{pmatrix}, Y =$$

$$\begin{pmatrix} 1 & 1 & 1 & a_1 & a_2 & a_n & \vdots & \vdots & a_1^{t-1} & a_2^{t-1} & a_n^{t-1} \end{pmatrix}, \text{ and } Z =$$

$$\begin{pmatrix} \frac{1}{g(a_1)} & 0 & 0 & 0 & \frac{1}{g(a_2)} & 0 & \cdots & 0 & 0 & \frac{1}{g(a_n)} \end{pmatrix},$$

Therefore, we have that any code word  $c \in \Gamma(L, g(z))$  if and only if  $Hc^T = 0$ .

#### 4.2.4 Encoding

Multiplying the input message by the Goppa Codes and generator matrix yields the encoded message.

**Definition:** The Goppa code generating matrix G is a k by n matrix with rows ordered according to their basis  $\Gamma(L, g(x))$ .

**Proposition:** In matrix theory, a generator matrix G is any a matrix with rank k for which



GHT = 0.

When we have to send the message we have to adopt the following steps:-

- Write the message in blocks of k Symbols
- The generator matrix (G) is then multiplied by each block of k symbols.
- The result we achieve after multiplying is denoted as code words.

Below is the example that explains the encoding of one block of message: -

$$(m_1, m_2, \dots, m_k) * G = (c_1, \dots, c_n).$$

#### 4.2.5 Irreducible Binary Goppa Code Example

As we have already explained that  $(2^4) \cong GF(2)[X]/(k(X))$  for each irreducible polynomial  $k(X)$  having a degree of 4. In the very first step we have to find the primitive element ( $\alpha$ ). By using mathematical formulas we can factor  $X^{15} - 1 \pmod{2}$  into irreducible factors [73].

$$X^{15} - 1 \pmod{2} = (X + 1)(X^2 + X + 1)(X^4 + X + 1)(X^4 + X^3 + 1)(X^4 + X^3 + X^2 + X + 1)$$

For example, if we imagine that  $k(X) = X^4 + X + 1$ , then a root of  $k(X)$  is assumed to be a primitive element if and only if the order of is 15. We just need to verify when 3 1 and 5 1 since we already know that 1 and that the order of an element should split the order of the group. By applying the equation  $\alpha^4 = \alpha + 1$ , we can compute  $\alpha^3 = \alpha^3 \neq 1$  and  $\alpha^5 = \alpha \cdot \alpha^4 = (1 + \alpha) = \alpha^2 + \alpha \neq 1$ .

We can conclude that,  $(2^4)^*$ , is the multiplicative group of all the nonzero elements which are present in  $GF(2^4)$ , also this is a cyclic subgroup that is generated by  $\alpha$ .

$$(2^4) = GF(2^4) \cup \{0\} = \{0, 1, \alpha, \alpha^2, \alpha^3, \dots, \alpha^{14}\}.$$

Therefore, we may represent the components of (24) using the powers of plus zero, much like binary notation. We rely on the fact that again.  $\alpha^4 = \alpha + 1$ .

$$0 = 0 \cdot 1 + 0 \cdot \alpha + 0 \cdot \alpha^2 + 0 \cdot \alpha^3 = (0, 0, 0, 0)^T$$

$$1 = 1 \cdot 1 + 0 \cdot \alpha + 0 \cdot \alpha^2 + 0 \cdot \alpha^3 = (1, 0, 0, 0)^T$$

$$\alpha = 0 \cdot 1 + 1 \cdot \alpha + 0 \cdot \alpha^2 + 0 \cdot \alpha^3 = (0, 1, 0, 0)^T$$

$$\alpha^2 = 0 \cdot 1 + 0 \cdot \alpha + 1 \cdot \alpha^2 + 0 \cdot \alpha^3 = (0, 0, 1, 0)^T$$

$$\alpha^3 = 0 \cdot 1 + 0 \cdot \alpha + 0 \cdot \alpha^2 + 1 \cdot \alpha^3 = (0, 0, 0, 1)^T$$

$$\alpha^4 = 1 \cdot 1 + 1 \cdot \alpha + 0 \cdot \alpha^2 + 0 \cdot \alpha^3 = (1, 1, 0, 0)^T$$

$$\alpha^5 = 0 \cdot 1 + 1 \cdot \alpha + 1 \cdot \alpha^2 + 0 \cdot \alpha^3 = (0, 1, 1, 0)^T$$

$$\alpha^6 = 0 \cdot 1 + 0 \cdot \alpha + 1 \cdot \alpha^2 + 1 \cdot \alpha^3 = (0,0,1,1)^T$$

$$\alpha^7 = 1 \cdot 1 + 1 \cdot \alpha + 0 \cdot \alpha^2 + 1 \cdot \alpha^3 = (1,1,0,1)^T$$

$$\alpha^8 = 1 \cdot 1 + 0 \cdot \alpha + 1 \cdot \alpha^2 + 0 \cdot \alpha^3 = (1,0,1,0)^T$$

$$\alpha^9 = 0 \cdot 1 + 1 \cdot \alpha + 0 \cdot \alpha^2 + 1 \cdot \alpha^3 = (0,1,0,1)^T$$

$$\alpha^{10} = 1 \cdot 1 + 1 \cdot \alpha + 1 \cdot \alpha^2 + 0 \cdot \alpha^3 = (1,1,1,0)^T$$

$$\alpha^{11} = 0 \cdot 1 + 1 \cdot \alpha + 1 \cdot \alpha^2 + 1 \cdot \alpha^3 = (0,1,1,1)^T$$

$$\alpha^{12} = 1 \cdot 1 + 1 \cdot \alpha + 1 \cdot \alpha^2 + 1 \cdot \alpha^3 = (1,1,1,1)^T$$

$$\alpha^{13} = 1 \cdot 1 + 0 \cdot \alpha + 1 \cdot \alpha^2 + 1 \cdot \alpha^3 = (1,0,1,1)^T$$

$$\alpha^{14} = 1 \cdot 1 + 0 \cdot \alpha + 0 \cdot \alpha^2 + 1 \cdot \alpha^3 = (1,0,0,1)^T$$

If we have Goppa code as  $L = \{\alpha^i \text{ such that } 2 \leq i \leq 13\}$  with  $(x) = x^2 + x + \alpha^3$ .

This code is supposed to be irreducible over  $(2^4)$ . However, this code has the parameters  $p = 2$ ,  $m = 4$ ,  $n = 12$ , and  $t = 2$ . We can conclude that  $k \geq n - mt = 12 - 4 \cdot 2 = 4$ .

As,  $d \geq 2t + 1 = 2 \cdot 2 + 1 = 5$ .

So, final Goppa Code would be  $[12, \geq 4, \geq 5]$

We can then compute  $XYZ =$

$H$ .

$$= (g_2 \cdot g(a_1^{-1}) g_2 \cdot g(a_2^{-1}) \cdots g_2 \cdot g(a_{12}^{-1})(g_1 + g_2 \cdot a_1) \cdot g(a_1^{-1})(g_1 + g_2 a_2) \cdot g(a_2^{-1}) \cdots g_1 + g_2 a_{12} \cdot g(a_{12}^{-1}))$$

$$= (a^3 \ a^9 \ a^4 \ a^1 \ a^8 a^6 \ a^3 \ a^6 \ a^1 \ a^2 \ a^2 \ a^8 \ 1 \ a^{13} \ a^7 \ a^{14} \ a^3 \ 0 \ a^{14} \ a^6 \ a^7 \ a^{10} \ a^4 \ a^{13})$$

$$=(0 \ 010100000010 \ 111000010000 \ 00011$$

$$0101111 \ 100011100001 \ 111001011110$$

$$010000011100 \ 100000101010 \ 11110111001)$$

Since,  $GH^T = 0$ . Therefore, we can easily calculate the rows of matrix  $G$  as the vectors of Null space ( $H \bmod 2$ )

Finally  $G$  will be equal to:

$$G = (011010100100011110011000 \ 11011$$

$$0000001111011010010)$$

We can see that this is a  $4 \times 12$  matrix which means that the dimension of  $\Gamma(L, g(x))$  is 4.

And also conclusively  $[12, 4, \geq 5]$  are the parameters of this Goppa code.

#### 4.2.6 Error Correction

We assume that the received code word is denoted as  $y$  with  $r \leq t$  errors. So,

$$y = (y_1, \dots, y_n) = (c_1, \dots, c_n) + (e_1, \dots, e_n),$$

This code word has  $r$  places in such a way where  $e_{ii} \neq 0$ . The mistake vector must be located before the corrected code words may be reinserted into the original message. To achieve this, one must first determine the set of error positions  $E = \{i \text{ such that } e_i \neq 0\}$  and the associated error values  $e_i$  for all  $i \in E$ .

**Definition 2.7.1** Error locating polynomial written as  $(x)$  can be defined as:

$$(x) = \prod_{i \in E} (x - \alpha_i)$$

In binary Goppa codes we can have only two possible values (Errors, No errors) so we only have

to find the location of the errors as all other values other than errors will be correct. But if we are using the regular conventional Goppa Code then along with the errors we also have to find out the error correction polynomial as well along with the location of errors[39].

We are using the Patterson's algorithm [H] in order to correct the errors present in the code word. The algorithm to find correct the errors in range of  $r \leq t$  for  $(x)$  which is irreducible over  $(2^m)$  is as follows:

1. Let  $y = (y_1, \dots, y_n)$  be a received code word. Compute the syndrome

$$s(x) = \sum_{i=1}^n \frac{y_i}{x - a_i} \text{ mod } g(x)$$

2. Below are steps which are needed to Calculate  $\sigma(x)$ :

- First step is to find  $h(x)$  such that  $s(x) h(x) \equiv 1 \pmod{g(x)}$ . If  $h(x) = x$ , then the answer is  $\sigma(x) = x$ .
- Second step is calculation of  $(x)$  such that  $d^2(x) \equiv h(x) + x \pmod{g(x)}$ .
- Third step is finding  $(x)$ , along with  $b(x)$  which is supposed to be of least degree, in such a way that  $d(x)b(x) \equiv (x) \pmod{g(x)}$ .
- Fourth and last step is to set  $(x) = a^2(x) + b^2(x)x$ .

3. Then, we utilize  $(x)$  to determine the set of error positions  $E = i$  for which  $(i) = 0$ .

4. The error vector  $e$  is defined in the following Step 4 as  $e_i = 1$  for  $i \in E$  and  $e_i = 0$  everywhere.

5. Finally, the secret phrase is defined as  $c = y.e$ .

### 4.2.7 Decoding

Original message may be readily recovered by the recipient after all possible faults in the code word have been fixed.

As we defined above  $(m_1, m_2, \dots, m_k) * G = (c_1, \dots, c_n)$ .

This equation can be rearranged as  $m_1, m_2, \dots, m_k * G = (c_1, \dots, c_n)$  so that  $G^T \cdot (m_1 \ m_2 \ \dots \ m_k) = (c_1 \ c_2 \ \dots \ c_n)$

Solving this equation by Row reduction method:

$$\begin{pmatrix} c_1 & c_2 & \dots & c_n \end{pmatrix} \sim \dots \sim \begin{pmatrix} 1 & 0 & \dots & 0 & m_1 & 0 & 1 & \dots & 0 & m_2 & \dots \\ \vdots & 0 & 0 & \dots & 1 & m_k & \dots & \dots & \dots & \dots & \dots \end{pmatrix} - - - - - X$$

Here, X is defined as a matrix of  $(n - k) \times (k + 1)$ .

### 4.4 The McEliece Cryptosystem with Example

Using a linear error-correcting code, the McEliece Cryptosystem generates public and private keys. The error-correcting binary Goppa code made its debut in the McEliece Cryptosystem. A public key is one that is freely accessible to the public. The public key is constructed using the public key, but in a manner that makes deconstructing it difficult. Send encrypted information to a specific recipient using a private key. I usually go back to the tale of Alice and Bob whenever I need to explain cryptography to someone[3].

Consider the hypothetical situation when Alice has something confidential to share with Bob. Bob must first share his public key with everyone before he may share his private key with anybody. After that, Alice uses Bob's public key to encrypt her message. Here, the information is coded into a hidden word. Even though it was encrypted, Bob was able to read her message. Now that Bob knows his codeword, he can decipher the message. He only has to use his hidden key[1].

To begin creating the public and private keys, Bob selects a Goppa polynomial (z) of degree t over GF (2m). The Goppa code values generated by the above-selected equation must fall between the range [n, mt- 2t+1]. Bob would then use these factors and the selected Goppa codes to calculate the Goppa code's generator matrix (G), which would be a k by n matrix. Next, Bob chooses two matrices, S (an invertible k by k matrix) and P (a permutation matrix n by n). Each row and column of (P) has the value 1, whereas all other cells have 0. After that, he calculates that  $G'=SGP$ .  $G'$ . The

solution to this equation is the single component of the public key,  $G'$ , together with the constant  $t$ . Bob's secret information includes the polynomial ( $z$ ), the original matrix ( $G$ ), and two additional matrices ( $S, P$ ) that are prepared so that  $G' = SGP[2]$ .

When Bob shares his public key, Alice generates a binary vector  $e$  at random, giving it a length of  $k$  and a weight of  $t$ . Alice may then encrypt her message  $m = (m_1, m_2, \dots, m_k)$  with the key  $y = mG' + e$ . The ciphertext  $y$  is then sent by Alice.

Bob learns Alice's code phrase and applies his knowledge of the permutation matrix  $P$  to the problem.

$$y' = yP^{-1} = mG'P^{-1} + eP^{-1} = mSGPP^{-1} + e' = (mS)G + e'$$

By employing Patterson's technique, Bob is able to decode  $y'$  into the message  $m' = mS$ . After this is complete, Bob is able to retrieve the original message by calculating  $m = m'S^{-1}$ . from  $y - e' = mSG$  because he already knows what  $S$  is.

### 4.3.1 Example

To explain the McEliece Cryptosystem in depth, we consider the generator matrix  $G$  as,

$$G = (01101010010001111001100011011000 \\ 0001111011010010)$$

In second step we have to select random matrices ( $S$ ) and ( $P$ ) and modify matrix ( $G$ ). As we can see that dimensions of  $G$  are  $4 \times 12$  so dimensions of  $S$  would also be  $4 \times 4$  and dimensions of  $P$  matrix would be  $12 \times 12$ . For this example we have selected the random matrix as:

$$S = (1001010101000011)$$

$$P = (10000000000100100000100$$

$$0.000000010111000001010000$$

$$0.00010001111010000001110000$$

$$1.0001010100000001100100000001$$

$$0.000000000000010000000000000100000$$



the matrix. Bob then correct these errors by using the error correcting algorithm which generates the following output:

$$mSG=(1,1,1,1,1,1,0,1,1,1,0).$$

In order to retrieve the original message m we reduce the given matrix by Row reduction procedure and calculate the final message as

$$mS = (1,1,0,1) \text{ and } m =$$

$$(1, 1, 0, 1) \cdot (1110001001110110) = (1, 0, 1, 0)$$



## Application Implementation and Results

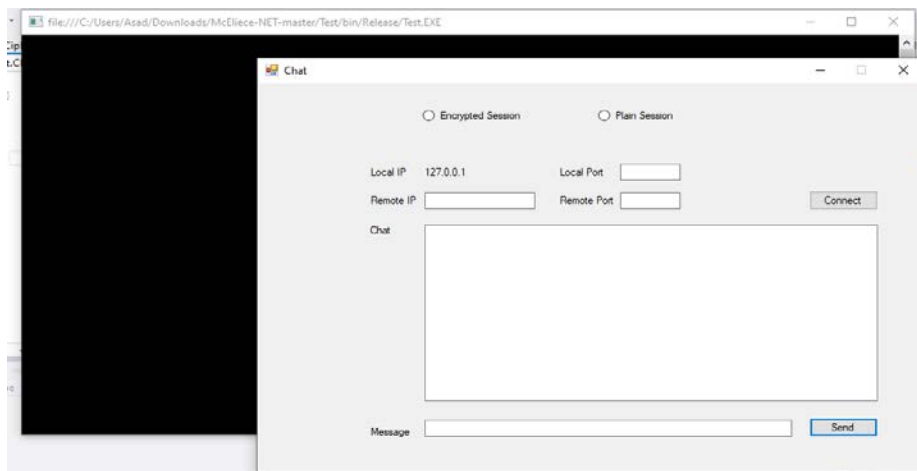
In order to develop the secure messaging system/ app based on McEliece cryptosystem we have used visual studio. We have divided the complete deployment process into multiple steps so that we have ease of understanding and debugging of each step of the communication model.

### 5.1 Application GUI

The GUI of application gives us two options of sending a text i.e. unencrypted messages and encrypted messages using McEliece Cryptosystems. There are two things of participating entities are required to enter in GUI before starting any chat session: -

- IP address
- Socket address

For debugging purpose, initially we have kept the static IP and Socket address for both the entities (fig 3) but these can be automatically assigned and kept in database.



**Figure 3: Initial Application GUI**

Above is the simple GUI that we have created for testing and debugging purpose. There are two options of starting a session i.e

- Encrypted Session
- Plain Session

If we select the Plain session then there will be no encryption taking place for sending of messages between the two parties. This feature is not necessary for the model but we have kept this for debugging purpose as to test whether two hosts are connected via socket programming or not. If we are able to send messages in plain mode, this confirms that two communicating parties are connected with each other. If we select the Encrypted session that means now messages will be sent after encryption with McEliece crypto.

Below the session choice we have local and remote IP and port address fields. Local IP and port address is of the sender Computer while the Remote IP and port address corresponds to the receiver computer. Then we have the chat and message fields where we write the messages that are to be sent to the receiver. Chat field is also used for debugging purpose during the deployment phase of the application. The plain message is written in the message box whereas cypher text corresponding to that plain text and reception acknowledgment of message on receiver end are displayed in the chat box.

Now we explain each step of implementation individually with the help of screenshots

## 5.2 Key Generation:

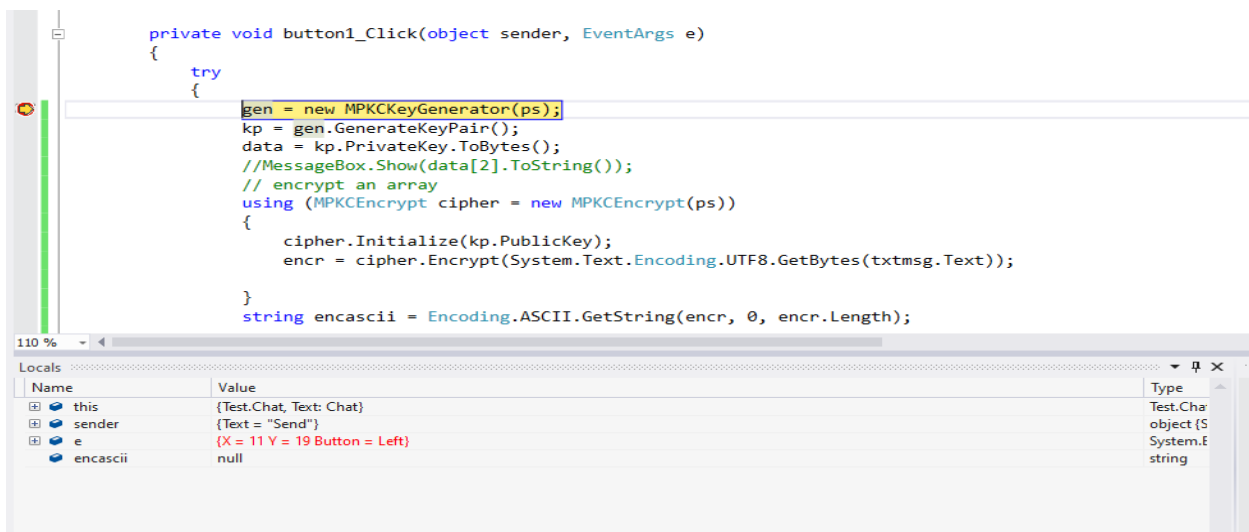


Figure 4: Key Generation Process

To begin encrypting and decrypting a message, we must first generate the corresponding public key and private key before proceeding to the next step. In order to do this, we first built a

function called "gen" that generates a set of both private and public encryption keys, as seen in figure(fig 9-12).

Using the publicly known parameters m, t, and n, the method creates a pair of keys. There are 3 parts to the public key: n, t, and G. Field polynomials, Goppa polynomials, G', S, and P are all part of the private key.

- If  $n = 2m$  and  $k$  is selected maximally, i.e.  $k = n - mt$ , then GC is a binary irreducible (n,k) Goppa code with up to  $t$  correctable mistakes.
- In the Goppa code GC, H is a parity check matrix of size  $mt$  by  $n$ .
- S is a  $k$  by  $k$  non-singular binary matrix with a random distribution.
- Here, P represents a random  $n$  by  $n$  permutation matrix.
- $G = SG'P$ .

For constructor class we have defined the following parameters (fig 5): -

The screenshot shows the Visual Studio IDE with the CTRPrng class constructor code. The code includes XML-style comments for parameters and a public constructor method. The constructor initializes several fields: \_engineType, \_seedType, \_byteBuffer, \_bufferSize, and \_keySize. A check is performed for BufferSize < 64, throwing an exception if true. The constructor is called with BlockEngine = RDX, SeedEngine = CSPRsg, BufferSize = 4096, and KeySize = 0.

```

#region Constructor
/// <summary>
/// Initialize the class
/// </summary>
///
/// <param name="BlockEngine">The block cipher that powers the rng (default is RDX)</param>
/// <param name="SeedEngine">The Seed engine used to create keyng material (default is CSPRsg)</param>
/// <param name="BufferSize">The size of the cache of random bytes (must be more than 1024 to enable parallel processing)</param>
/// <param name="KeySize">The key size (in bytes) of the symmetric cipher; a <0</0> value will auto size the key</param>
public CTRPrng(BlockCiphers BlockEngine = BlockCiphers.RDX, SeedGenerators SeedEngine = SeedGenerators.CSPRsg, int BufferSize = 4096, int KeySize = 0)
{
    if (BufferSize < 64)
        throw new CryptoRandomException("CTRPrng:Ctor", "Buffer size must be at least 64 bytes!", new ArgumentNullException());

    _engineType = BlockEngine;
    _seedType = SeedEngine;
    _byteBuffer = new byte[BufferSize];
    _bufferSize = BufferSize;
    if (KeySize > 0)
        _keySize = KeySize;
    else
        _keySize = GetKeySize(BlockEngine);
}

```

The Locals window shows the following variables and their values:

Name	Value	Type
this	{VTDev.Libraries.CEXEngine.Crypto.Prng.CTRPrng}	VTDev.Li
BlockEngine	RDX	VTDev.Li
SeedEngine	CSPRsg	VTDev.Li
BufferSize	4096	int
KeySize	0	int

Figure 5: Initial Parameters Generation

**BlockEngine:** The block cipher that powers the rng (default is RDX)

**SeedEngine:** The Seed engine used to create keyng material (default is CSPRsg)

**BufferSize:** The size of the cache of random bytes (must be more than 1024 to enable parallel processing)

**KeySize:** The key size (in bytes) of the symmetric cipher; a value will auto size the key

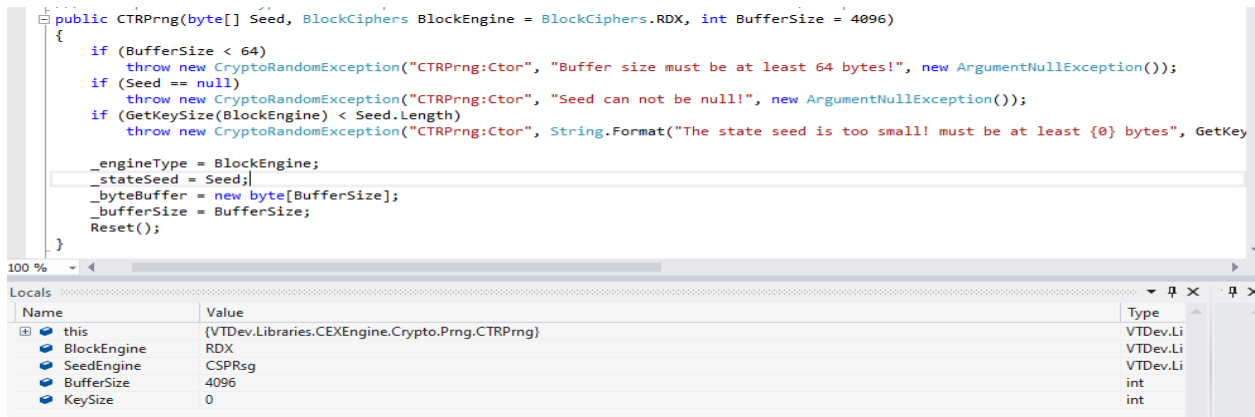


Figure 6: Initial Parameters Created (1)

**CipherParams:** The RLWE Parameters instance containing the cipher settings

**Rng Engine:** An initialized Prng instance

**Parallel:** Use parallel processing when generating a key; set to false if using a passphrase type generator (default is true)

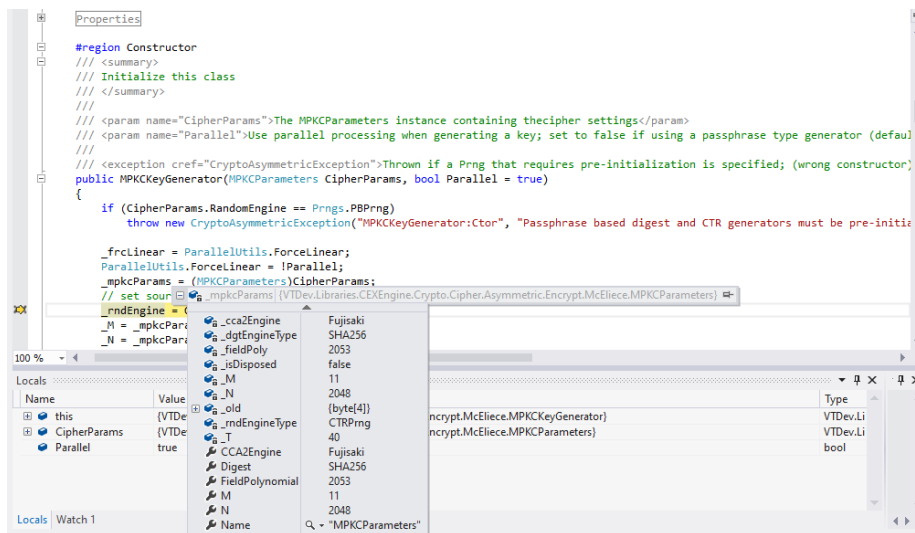


Figure 7: Initial Parameters Created (2)

Above mentioned parameters are generated after series of iterations (fig 6-7).

After defining the above-mentioned parameters next step is generation of finite field  $GF(2^m)$  with fixed field polynomial as shown in fig 8-9

```

        this._degree = Degree;
        _polynomial = PolynomialRingGF2.GetIrreduciblePolynomial(Degree);
    }

    /// <summary>
    /// Create a finite field GF(2^m) with the fixed field polynomial
    /// </summary>
    ///
    /// <param name="Degree">The degree of the field</param>
    /// <param name="Polynomial">The field polynomial</param>
    public GF2mField(int Degree, int Polynomial)
    {
        if (Degree != PolynomialRingGF2.Degree(Polynomial))
            throw new ArgumentException(" Error: the degree is not correct!");
        if (!PolynomialRingGF2.IsIrreducible(Polynomial))
            throw new ArgumentException(" Error: given polynomial is reducible!");

        _degree = Degree;
        _polynomial = Polynomial;
    }

    /// <summary>

```

Name	Value	Type
this	{Finite Field GF(2^0) = GF(2)[X]/<0> }	VTDev.Li
Degree	11	int
Polynomial	2053	int

Figure 8: Finite Field Generation (1)

```

    }

    /// <summary>
    /// Create a finite field GF(2^m) with the fixed field polynomial
    /// </summary>
    ///
    /// <param name="Degree">The degree of the field</param>
    /// <param name="Polynomial">The field polynomial</param>
    public GF2mField(int Degree, int Polynomial)
    {
        if (Degree != PolynomialRingGF2.Degree(Polynomial))
            throw new ArgumentException(" Error: the degree is not correct!");
        if (!PolynomialRingGF2.IsIrreducible(Polynomial))
            throw new ArgumentException(" Error: given polynomial is reducible!");

        _degree = Degree;
        _polynomial = Polynomial;
    }
}

```

Name	Value	Type
this	{Finite Field GF(2^0) = GF(2)[X]/<0> }	VTDev.Li
_degree	0	int
_polynomial	0	int
Degree	0	int
Polynomial	0	int
Degree	11	int
Polynomial	2053	int

Figure 9: Finite Field Generation (2)

Methods for working with polynomials over the finite field GF(2) are described in "this" class., In next step we have checked the irreducibility of polynomial as it must be irreducible for generation of keys.

```

    /// <summary>
    /// Checking polynomial for irreducibility
    /// </summary>
    ///
    /// <param name="P">The polinomial</param>
    ///
    /// <returns>Returns true if p is irreducible and false otherwise</returns>
    public static bool IsIrreducible(int P)
    {
        if (P == 0)
            return false;
        int d = IntUtils.URShift(Degree(P), 1);
        int u = 2;

        for (int i = 0; i < d; i++)
        {
            u = ModMultiply(u, u, P);
            if (Gcd(u ^ 2, P) != 1)
                return false;
        }

        return true;
    }
    /// <summary>

```

Name	Value	Type
P	2053	int
d	0	int
u	0	int

Figure 10: Checking Reducibility of Finite Field (1)

```

    /// <summary>
    /// Create a finite field GF(2^m) with the fixed field polynomial
    /// </summary>
    ///
    /// <param name="Degree">The degree of the field</param>
    /// <param name="Polynomial">The field polynomial</param>
    public GF2mField(int Degree, int Polynomial)
    {
        if (Degree != PolynomialRingGF2.Degree(Polynomial))
            throw new ArgumentException(" Error: the degree is not correct!");
        if (!PolynomialRingGF2.IsIrreducible(Polynomial))
            throw new ArgumentException(" Error: given polynomial is reducible!");

        _degree = Degree;
        _polynomial = Polynomial;
    }

    /// <summary>
    /// Create a finite field GF(2^m) using an encoded array
    /// </summary>
    ///
    /// <param name="Encoded">The polynomial and degree encoded as a byte array</param>
    public GF2mField(byte[] Encoded)

```

Name	Value	Type
this	{Finite Field GF(2^0) = GF(2)[X]/<0> }	VTDev.Libraries.CEXEngine.Cipher.Asymmetric.Encrypt.McEliece.Algebra.GF2mField
Degree	11	int
Polynomial	2053	int

Figure 11: Checking Reducibility of Finite Field (2)

Next phase is generation of irreducible Goppa Polynomial which is further used for generation of matrixes and goppa codes

```

#region public methods
/// <summary>
/// Generate an encryption key pair
/// </summary>
///
/// <returns>A McElieceKeyPair containing public and private keys</returns>
public IAsymmetricKeyPair GenerateKeyPair()
{
    // finite field GF(2^m)
    GF2mField field = new GF2mField(_M, _fieldPoly);
    // irreducible Goppa polynomial
    PolynomialGF2mSmallM gp = new PolynomialGF2mSmallM(field, _T, PolynomialGF2mSmallM.RANDOM_IRREDUCIBLE_POLYNOMIAL, _rndEngine);
    PolynomialRingGF2m ring = new PolynomialRingGF2m(field, gp);
    // matrix for computing square roots in (GF(2^m))^t
    PolynomialGF2mSmallM[] qInv = ring.SquareRootMatrix();
    // generate canonical check matrix
    GF2Matrix h = GoppaCode.CreateCanonicalCheckMatrix(field, gp);
    // compute short systematic form of check matrix
    GoppaCode.MaMaPe mmp = GoppaCode.ComputeSystematicForm(h, _rndEngine);
    GF2Matrix shortH = mmp.SecondMatrix();
    Permutation p = mmp.Permutation();
    // compute short systematic form of generator matrix
    GF2Matrix shortG = (GF2Matrix)shortH.ComputeTranspose();
    // obtain number of rows of G (= dimension of the code)
}

```

Figure 12: Goppa Polynomial Generation (1)

Name	Value	Type
isDisposed	false	bool
M	11	int
mpkcParams	{VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.MPKCParameters}	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.MPKCParameters
N	2048	int
rndEngine	{VTDev.Libraries.CEXEngine.Crypto.Prng.CTRPrng}	VTDev.Libraries.CEXEngine.Crypto.Prng.Random {VTDev.Libraries.CEXEngine.Crypto.Prng.CTRPrng}
T	40	int
Name	"MPKCKeyGenerator"	string
Static members		
field	{Finite Field GF(2^11) = GF(2)[X]/<1+x^2+x^11>}	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.Algebra.GF2mField
gp	{ Polynomial over Finite Field GF(2^11) = GF(2)[X]/<1+x^2+x^11> : 0111000011 }	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.Algebra.PolynomialGF2mS
coefficients	{int[41]}	int[]
degree	40	int
field	{Finite Field GF(2^11) = GF(2)[X]/<1+x^2+x^11>}	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.Algebra.GF2mField
Degree	40	int
Field	{Finite Field GF(2^11) = GF(2)[X]/<1+x^2+x^11>}	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.Algebra.GF2mField
Head	1	int
Static members		
ring	null	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.Algebra.PolynomialRingGF
qInv	null	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.Algebra.PolynomialGF2mS
h	null	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.Algebra.GF2Matrix
mmp	null	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.Algebra.GoppaCode.MaMaPe
shortH	null	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.Algebra.GF2Matrix
p	null	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.Algebra.Permutation
shortG	null	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.Algebra.GF2Matrix
k	0	int
pubKey	null	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Interfaces.IAsymmetricKey
privKey	null	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Interfaces.IAsymmetricKey

Figure 13: Goppa Polynomial Generation (2)

Goppa Polynomial Coefficients with degree 40 are shown in fig 14-16

Name	Value	Type
coefficients	[int[41]]	int[]
[0]	902	int
[1]	15	int
[2]	1761	int
[3]	457	int
[4]	1982	int
[5]	355	int
[6]	695	int
[7]	575	int
[8]	123	int
[9]	206	int
[10]	1378	int
[11]	587	int
[12]	1488	int
[13]	1398	int
[14]	1307	int
[15]	1973	int
[16]	1701	int
[17]	1614	int
[18]	1602	int
[19]	1827	int
[20]	372	int
[21]	574	int
[22]	837	int
[23]	1324	int
[24]	277	int
[25]	327	int

Figure 14: Goppa Polynomial (1)

[16]	1701	int
[17]	1614	int
[18]	1602	int
[19]	1827	int
[20]	372	int
[21]	574	int
[22]	837	int
[23]	1324	int
[24]	277	int
[25]	327	int
[26]	1855	int
[27]	524	int
[28]	453	int
[29]	1453	int
[30]	51	int
[31]	1702	int
[32]	739	int
[33]	515	int
[34]	977	int
[35]	1546	int
[36]	803	int
[37]	300	int
[38]	113	int
[39]	1885	int
[40]	1	int
_degree	40	int

Figure 15: Goppa Polynomial (2)

Then in next step we have generated canonical check matrix H and systematic form of check matrix by using goppa polynomial as shown in fig 16



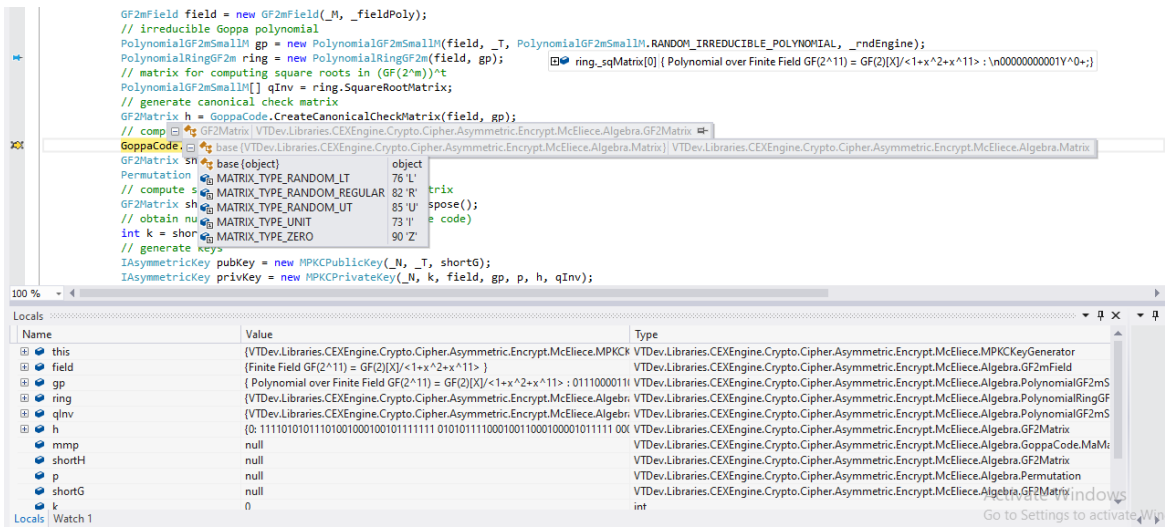
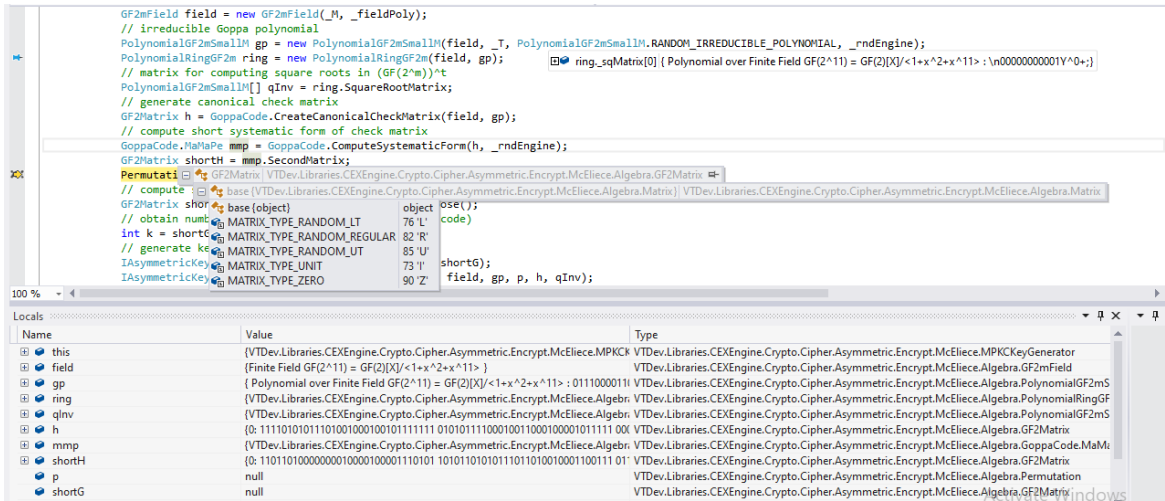


Figure 16: Generation of Canonical Matrix



Result/Output (H matrix)

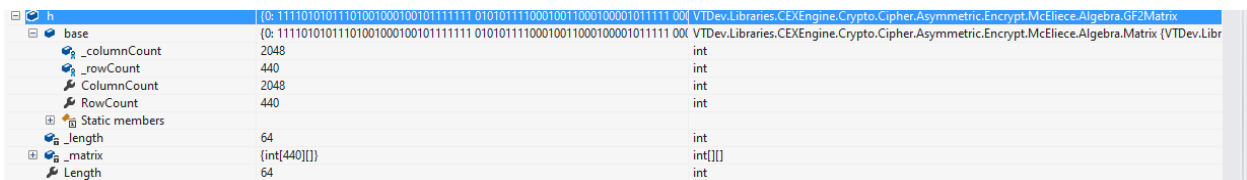


Figure 17: Generation of H Matrix

Name	Value	Type
h_matrix	(int[440])	int[]
[0]	(int[64])	int[]
[1]	(int[64])	int[]
[2]	(int[64])	int[]
[3]	(int[64])	int[]
[4]	(int[64])	int[]
[5]	(int[64])	int[]
[6]	(int[64])	int[]
[7]	(int[64])	int[]
[8]	(int[64])	int[]
[9]	(int[64])	int[]
[10]	(int[64])	int[]
[11]	(int[64])	int[]
[12]	(int[64])	int[]
[13]	(int[64])	int[]
[14]	(int[64])	int[]
[15]	(int[64])	int[]
[16]	(int[64])	int[]
[17]	(int[64])	int[]
[18]	(int[64])	int[]
[19]	(int[64])	int[]
[20]	(int[64])	int[]

Figure 18: H Matrix (1)

Name	Value	Type
[421]	(int[64])	int[]
[422]	(int[64])	int[]
[423]	(int[64])	int[]
[424]	(int[64])	int[]
[425]	(int[64])	int[]
[426]	(int[64])	int[]
[427]	(int[64])	int[]
[428]	(int[64])	int[]
[429]	(int[64])	int[]
[430]	(int[64])	int[]
[431]	(int[64])	int[]
[432]	(int[64])	int[]
[433]	(int[64])	int[]
[434]	(int[64])	int[]
[435]	(int[64])	int[]
[436]	(int[64])	int[]
[437]	(int[64])	int[]
[438]	(int[64])	int[]
[439]	(int[64])	int[]

Figure 19: H Matrix (2)

Name	Value	Type
[439]	(int[64])	int[]
[0]	-983706010	int
[1]	91300953	int
[2]	1548435510	int
[3]	1473606004	int
[4]	1859187636	int
[5]	968063565	int
[6]	-1395651542	int
[7]	-1981064605	int
[8]	1789527735	int
[9]	-656849560	int
[10]	40883043	int
[11]	496547310	int
[12]	-948085446	int
[13]	-1563082722	int
[14]	1400903774	int
[15]	2082078226	int
[16]	2085512406	int
[17]	1037782683	int
[18]	2038896285	int
[19]	1601973358	int
[20]	-1158084103	int

Figure 20: H Matrix (3)

After generation of H matrix we have generated G, P and S matrixes which are further used for keys generation as shown in fig 21-23

## -G matrix

shortG	{0: 11010010110111111010110011100011 001110001010101111110111011010 11( VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.Algebra.GF2Matrix	
base	{0: 11010010110111111010110011100011 001110001010101111110111011010 11( VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.Algebra.Matrix {VTDev.Libr	
_columnCount	440	int
_rowCount	1608	int
ColumnCount	440	int
RowCount	1608	int
Static members		
MATRIX_TYPE_RANDOM_LT	76 'L'	char
MATRIX_TYPE_RANDOM_REG	82 'R'	char
MATRIX_TYPE_RANDOM_UT	85 'U'	char
MATRIX_TYPE_UNIT	73 'I'	char
MATRIX_TYPE_ZERO	90 'Z'	char
_length	14	int
_matrix	{int[1608][]}	int[[]]
Length	14	int

Figure 21: Generator Matrix

Matrix has 440 columns with 1608 rows.

Locals		
Name	Value	Type
_matrix	{int[1608][]}	int[[]]
[0]	{int[14]}	int[]
[1]	{int[14]}	int[]
[2]	{int[14]}	int[]
[3]	{int[14]}	int[]
[4]	{int[14]}	int[]
[5]	{int[14]}	int[]
[6]	{int[14]}	int[]
[7]	{int[14]}	int[]
[8]	{int[14]}	int[]
[9]	{int[14]}	int[]
[10]	{int[14]}	int[]
[11]	{int[14]}	int[]
[12]	{int[14]}	int[]
[13]	{int[14]}	int[]
[14]	{int[14]}	int[]
[15]	{int[14]}	int[]
[16]	{int[14]}	int[]
[17]	{int[14]}	int[]
[18]	{int[14]}	int[]
[19]	{int[14]}	int[]
[20]	{int[14]}	int[]
[21]	{int[14]}	int[]
[22]	{int[14]}	int[]
[23]	{int[14]}	int[]

Locals		
Name	Value	Type
+ [1581]	{int[14]}	int[]
+ [1582]	{int[14]}	int[]
+ [1583]	{int[14]}	int[]
+ [1584]	{int[14]}	int[]
+ [1585]	{int[14]}	int[]
+ [1586]	{int[14]}	int[]
+ [1587]	{int[14]}	int[]
+ [1588]	{int[14]}	int[]
+ [1589]	{int[14]}	int[]
+ [1590]	{int[14]}	int[]
+ [1591]	{int[14]}	int[]
+ [1592]	{int[14]}	int[]
+ [1593]	{int[14]}	int[]
+ [1594]	{int[14]}	int[]
+ [1595]	{int[14]}	int[]
+ [1596]	{int[14]}	int[]
+ [1597]	{int[14]}	int[]
+ [1598]	{int[14]}	int[]
+ [1599]	{int[14]}	int[]
+ [1600]	{int[14]}	int[]
+ [1601]	{int[14]}	int[]
+ [1602]	{int[14]}	int[]
+ [1603]	{int[14]}	int[]
+ [1604]	{int[14]}	int[]
+ [1605]	{int[14]}	int[]
+ [1606]	{int[14]}	int[]
+ [1607]	{int[14]}	int[]
Length	14	int

Each entry has further 14 sub values as shown below:-

+ [1606]	{int[14]}	int[]
+ [1607]	{int[14]}	int[]
[0]	1016407633	int
[1]	-1529248160	int
[2]	1200340282	int
[3]	-141519750	int
[4]	240234500	int
[5]	1177248467	int
[6]	1570458784	int
[7]	39350629	int
[8]	301296606	int
[9]	174851979	int
[10]	-167627595	int
[11]	382661601	int
[12]	95360482	int
[13]	16179108	int
Length	14	int

### -Permutation Matrix P

_p	[[1568, 420, 1446, 1184, 1301, 849, 321, 860, 87, 1640, 2029, 731, 595, 600, 1358, 991]	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.Algebra.Permutation
_perm	{int[2048]}	int[]
_s	{0: 01000101110100100010011001100101 0010011011110011101110110011000 01}	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.Algebra.GF2Matrix
FirstMatrix	{0: 01000101110100100010011001100101 0010011011110011101110110011000 01}	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.Algebra.GF2Matrix
Permutation	[[1568, 420, 1446, 1184, 1301, 849, 321, 860, 87, 1640, 2029, 731, 595, 600, 1358, 991]	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.Algebra.Permutation
SecondMatrix	{0: 11011010000000010000100001110101 10101101010111011010010001100111 01}	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.Algebra.GF2Matrix

Figure 22: Permutation Matrix (1)

Total of 2048 entries with random values as shown below:-

perm	[int[2048]]	int[]
[0]	1568	int
[1]	420	int
[2]	1446	int
[3]	1184	int
[4]	1301	int
[5]	849	int
[6]	321	int
[7]	860	int
[8]	87	int
[9]	1640	int
[10]	2029	int
[11]	731	int
[12]	595	int
[13]	600	int
[14]	1358	int
[15]	991	int
[16]	232	int
[17]	2022	int
[18]	1764	int
[19]	941	int
[20]	1023	int
[21]	1609	int
[22]	1367	int
[23]	217	int
[24]	551	int
[25]	1793	int
[26]	2046	int

Activat

[2022]	1289	int
[2023]	231	int
[2024]	1839	int
[2025]	753	int
[2026]	1913	int
[2027]	928	int
[2028]	865	int
[2029]	612	int
[2030]	38	int
[2031]	856	int
[2032]	1694	int
[2033]	956	int
[2034]	1672	int
[2035]	1110	int
[2036]	1174	int
[2037]	855	int
[2038]	1405	int
[2039]	969	int
[2040]	1103	int
[2041]	1114	int
[2042]	1903	int
[2043]	1305	int
[2044]	839	int
[2045]	1719	int
[2046]	1158	int
[2047]	933	int

Figure 23: Permutation Matrix (1-2)

Results achieved after first permutation are shown in fig 24:-

FirstMatrix	{0: 01000101110100100010011001100101 0010011011110011101110110011000 01: VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.Algebra.GF2Matrix
base	{0: 01000101110100100010011001100101 0010011011110011101110110011000 01: VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.Algebra.Matrix (VTDev.Libr
base_length	14 int
base_matrix	{int[440][[]]} int[[]]
Length	14 int

FirstMatrix	{0: 01000101110100100010011001100101 001001101111001101110110011000 01}	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.Algebra.GF2Matrix
base	{0: 01000101110100100010011001100101 001001101111001101110110011000 01}	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.Algebra.Matrix (VTDev.Libr
_columnCount	440	int
_rowCount	440	int
ColumnCount	440	int
RowCount	440	int
Static members		
MATRIX_TYPE_RANDOM_L	76 'L'	char
MATRIX_TYPE_RANDOM_F	82 'R'	char
MATRIX_TYPE_RANDOM_L	85 'U'	char
MATRIX_TYPE_UNIT	73 'I'	char
MATRIX_TYPE_ZERO	90 'Z'	char
_length	14	int
_matrix	{int[440][]}	int[[]]
Length	14	int

Figure 24: Results after first permutation

It has total 440 rows and 440 Columns as shown below

_matrix	{int[440][]}	int[[]]
[0]	{int[14]}	int[]
[1]	{int[14]}	int[]
[2]	{int[14]}	int[]
[3]	{int[14]}	int[]
[4]	{int[14]}	int[]
[5]	{int[14]}	int[]
[6]	{int[14]}	int[]
[7]	{int[14]}	int[]
[8]	{int[14]}	int[]
[9]	{int[14]}	int[]
[10]	{int[14]}	int[]
[11]	{int[14]}	int[]
[12]	{int[14]}	int[]
[13]	{int[14]}	int[]
[14]	{int[14]}	int[]
[15]	{int[14]}	int[]
[16]	{int[14]}	int[]
[17]	{int[14]}	int[]
[18]	{int[14]}	int[]
[19]	{int[14]}	int[]
[20]	{int[14]}	int[]
[21]	{int[14]}	int[]
[22]	{int[14]}	int[]
[23]	{int[14]}	int[]
[24]	{int[14]}	int[]
[25]	{int[14]}	int[]
[26]	{int[14]}	int[]

[414]	{int[14]}	int[]
[415]	{int[14]}	int[]
[416]	{int[14]}	int[]
[417]	{int[14]}	int[]
[418]	{int[14]}	int[]
[419]	{int[14]}	int[]
[420]	{int[14]}	int[]
[421]	{int[14]}	int[]
[422]	{int[14]}	int[]
[423]	{int[14]}	int[]
[424]	{int[14]}	int[]
[425]	{int[14]}	int[]
[426]	{int[14]}	int[]
[427]	{int[14]}	int[]
[428]	{int[14]}	int[]
[429]	{int[14]}	int[]
[430]	{int[14]}	int[]
[431]	{int[14]}	int[]
[432]	{int[14]}	int[]
[433]	{int[14]}	int[]
[434]	{int[14]}	int[]
[435]	{int[14]}	int[]
[436]	{int[14]}	int[]
[437]	{int[14]}	int[]
[438]	{int[14]}	int[]
[439]	{int[14]}	int[]

Figure 25: Number of columns after permutation

Second matrix generated after the second permutation have 1608 columns, 440 Rows and each values have 51 entries as shown in fig 26-29

SecondMatrix	{0: 11011010000000010000100001110101 10101101010111011010010001100111 01 VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.Algebra.GF2Matrix	
base	{0: 11011010000000010000100001110101 10101101010111011010010001100111 01 VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.Algebra.Matrix [VTDev.Libr	
_columnCount	1608	int
_rowCount	440	int
ColumnCount	1608	int
RowCount	440	int
Static members		
MATRIX_TYPE_RANDOM_L	76 'L'	char
MATRIX_TYPE_RANDOM_F	82 'R'	char
MATRIX_TYPE_RANDOM_U	85 'U'	char
MATRIX_TYPE_UNIT	73 'I'	char
MATRIX_TYPE_ZERO	90 'Z'	char
_length	51	int
_matrix	{int[440][]}	int[][]
Length	51	int

Figure 26: Second Matrix (1)

Name	Value	Type
matrix	{int[440]}	int[]
[0]	{int[51]}	int[]
[1]	{int[51]}	int[]
[2]	{int[51]}	int[]
[3]	{int[51]}	int[]
[4]	{int[51]}	int[]
[5]	{int[51]}	int[]
[6]	{int[51]}	int[]
[7]	{int[51]}	int[]
[8]	{int[51]}	int[]
[9]	{int[51]}	int[]
[10]	{int[51]}	int[]
[11]	{int[51]}	int[]
[12]	{int[51]}	int[]
[13]	{int[51]}	int[]
[14]	{int[51]}	int[]
[15]	{int[51]}	int[]
[16]	{int[51]}	int[]
[17]	{int[51]}	int[]
[18]	{int[51]}	int[]
[19]	{int[51]}	int[]
[20]	{int[51]}	int[]
[21]	{int[51]}	int[]
[22]	{int[51]}	int[]
[23]	{int[51]}	int[]
[24]	{int[51]}	int[]
[25]	{int[51]}	int[]
[26]	{int[51]}	int[]

Figure 27: Second Matrix (2)

Name	Value	Type
[414]	{int[51]}	int[]
[415]	{int[51]}	int[]
[416]	{int[51]}	int[]
[417]	{int[51]}	int[]
[418]	{int[51]}	int[]
[419]	{int[51]}	int[]
[420]	{int[51]}	int[]
[421]	{int[51]}	int[]
[422]	{int[51]}	int[]
[423]	{int[51]}	int[]
[424]	{int[51]}	int[]
[425]	{int[51]}	int[]
[426]	{int[51]}	int[]
[427]	{int[51]}	int[]
[428]	{int[51]}	int[]
[429]	{int[51]}	int[]
[430]	{int[51]}	int[]
[431]	{int[51]}	int[]
[432]	{int[51]}	int[]
[433]	{int[51]}	int[]
[434]	{int[51]}	int[]
[435]	{int[51]}	int[]
[436]	{int[51]}	int[]
[437]	{int[51]}	int[]
[438]	{int[51]}	int[]
[439]	{int[51]}	int[]
Length	51	int

Figure 28: Second Matrix (3)



Name	Value	Type
[439]	[int[51]]	int[]
[0]	866663350	int
[1]	-790191687	int
[2]	-1481137443	int
[3]	2147183474	int
[4]	-285762224	int
[5]	-1505178567	int
[6]	-2034024217	int
[7]	1453142562	int
[8]	-1839735133	int
[9]	-1454479814	int
[10]	1492253857	int
[11]	630488381	int
[12]	406941696	int
[13]	-553876399	int
[14]	-1577562689	int
[15]	263703014	int
[16]	-83940970	int
[17]	627791006	int
[18]	-1303326472	int
[19]	-1298326757	int
[20]	-2044602998	int
[21]	1772606460	int
[22]	-1188295553	int
[23]	-1553043773	int
[24]	1164893754	int
[25]	-894695852	int

Figure 29: Second Matrix (4)

Value of “k” generated at the end of the generation of all matrices is 1608 as shown in fig 30.

Name	Value	Type
this	{VTDDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.MPKCK	VTDDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.MPKKeyGenerator
field	{Finite Field GF(2^11) = GF(2)[X]/<1+x^2+x^11> }	VTDDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.Algebra.GF2mField
gp	{ Polynomial over Finite Field GF(2^11) = GF(2)[X]/<1+x^2+x^11> : 011100011	VTDDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.Algebra.PolynomialGF2mS
ring	{VTDDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.Algebr	VTDDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.Algebra.PolynomialRingGF
qInv	{VTDDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.Algebr	VTDDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.Algebra.PolynomialGF2mS
h	{0: 111101010111010010001001011111 010101111000100110001000101111 00	VTDDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.Algebra.GF2Matrix
mmp	{VTDDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.Algebr	VTDDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.Algebra.GoppaCode.MaM
shortH	{0: 1101101000000010000100001110101 1010110101011101101001000110011 01	VTDDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.Algebra.GF2Matrix
p	{11568, 420, 1446, 1184, 1301, 849, 321, 860, 87, 1640, 2029, 731, 595, 600, 1358, 991	VTDDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.Algebra.Permutation
shortG	{0: 1101001011011111010110011100011 0011100010101011111010111011010 11	VTDDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.Algebra.GF2Matrix
k	1608	int
pubKey	null	VTDDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Interfaces.IAsymmetricKey
privKey	null	VTDDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Interfaces.IAsymmetricKey

Figure 30: K value

Up till this process generation of all Matrices is completed. With the help of these generated matrices, we will create pair of keys and encrypt the messages to be sent.

### 5.3 Keys Generation

Basic algorithm for generation of Ordered pair of keys is below: -

Alice chooses a linear code C from a family of codes for which she already knows a good way to decode it. She then makes C public but keeps the method for decoding it secret. To use this kind of decoding method, you need to know the parameters used to define the chosen family of codes and also understand C, in the sense that you know how to make a random generating matrix.

For binary Goppa codes, examples of such data are the Goppa polynomial and error locators. So, Alice could release a C generator matrix that has been covered up in the right way.

Here are the specific steps you need to take:

- Alice picks a code C from a big family of codes, such as binary Goppa codes, that can (efficiently) fix  $t$  mistakes. An effective decoding algorithm, A, should emerge from this selection. Assume further that C has a generator matrix G. There is a wide variety of possible generating matrices for linear codes, but often one stands out as the best. This information should be kept hidden since it reveals A.
- To begin, Alice chooses a binary non-singular matrix S of size  $k$  by  $k$  at random.
- A random  $n \times n$  permutation matrix P is chosen by Alice.
- Alice determines  $G = SGP$  to be a  $k \times n$  matrix.
- $(N, G, t)$  is Alice's public key, and  $(S, P, A)$  is her private key.

where

- "N">The length of the code
- "T">The error correction capability of the code
- "G"> Generator matrix

```

    /// </summary>
    ///
    /// <param name="N">The length of the code</param>
    /// <param name="T">The error correction capability of the code</param>
    /// <param name="G">The generator matrix</param>
    internal MPKCPublicKey(int N, int T, GF2Matrix G)
    {
        _N = N;
        _T = T;
        _G = new GF2Matrix(G);
    }
    /// <summary>
    /// Constructor used by McElieceKeyFactory
  
```

Name	Value	T.
this	{VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.MPKCPublicKey}	V
N	2048	in
T	40	in
G	{0: 110101101111111111111011100110111 10010000010010001011111001100010 10001001101010111010011010000001 100001101111101001010101010: V	V
base	{0: 110101101111111111111011100110111 10010000010010001011111001100010 10001001101010111010011010000001 100001101111101001010101010: V	V
_length	14	in
_matrix	{int[1608][]}	in
Length	14	in

Figure 31: Initial Parameters Created

G	{0: 11010110111111111111011100110111 10010000010010001011111001100010 10001001101010111010011010000001 1000011011111010010101010: VTDev.Li	
base	{0: 11010110111111111111011100110111 10010000010010001011111001100010 10001001101010111010011010000001 1000011011111010010101010: VTDev.Li	
_columnCount	440	int
_rowCount	1608	int
ColumnCount	440	int
RowCount	1608	int
Static members		
MATRIX_TYPE_RANDOM_LT	76 'L'	char
MATRIX_TYPE_RANDOM_REGULAR	82 'R'	char
MATRIX_TYPE_RANDOM_UT	85 'U'	char
MATRIX_TYPE_UNIT	73 'I'	char
MATRIX_TYPE_ZERO	90 'Z'	char
_length	14	int

Figure 32: Length of rows and Columns

Length of the Matrix is 1607 with 14 sub entries

Public key generated by using above parameters and matrixes is as shown in fig 33

```

// compute short systematic form of check matrix
GoppaCode.MaMaPe mmp = GoppaCode.ComputeSystematicForm(h, rndEngine);
GF2Matrix shortH = mmp.SecondMatrix;
Permutation p = mmp.Permutation;
// compute short systematic form of generator matrix
GF2Matrix shortG = (GF2Matrix)shortH.ComputeTranspose();
// obtain number of rows of G (= dimension of the code)

```

Name	Value	Type
gp	{ Polynomial over Finite Field GF(2^11) = GF(2)[X]/<1-x^2+x^11> : 00011000000Y^0+1110000001	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Er
ring	{ VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.Algebra.PolynomialRing	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Er
qInv	{ VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.Algebra.PolynomialGF2	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Er
h	{ 0: 0010001000111101101100100001000 010001100000000001001110110011 00111100100011011C	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Er
mmp	{ VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.Algebra.GoppaCode.Ma	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Er
shortH	{ 0: 11111100000011110001001111010010 101000101001010100010100000111101 00111010001000010C	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Er
p	{ {1800, 1134, 317, 623, 223, 1913, 1921, 573, 607, 1097, 935, 901, 859, 633, 585, 979, 1379, 1068, 1160,	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Er
shortG	{ 0: 11010110111111111111011100110111 10010000010010001011111001100010 10001001101010111C	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Er
k	1608	int
pubKey	{ VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.MPKCPublicKey}	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.In
VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.MPKCPublicKey	{ VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.MPKCPublicKey}	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Er
_G	{ 0: 11010110111111111111011100110111 10010000010010001011111001100010 10001001101010111C	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Er
_isDisposed	false	bool
_N	2048	int
_T	40	int
G	{ 0: 11010110111111111111011100110111 10010000010010001011111001100010 10001001101010111C	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Er
K	1608	int
N	2048	int
Name	"MPKCPublicKey"	string
T	40	int
Static members		
privKey	null	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.In

Figure 33: Public Key (1)

Name	Value	Type
pubKey	{ VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.MPKCPublicKey}	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.In
VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.MPKCPublicKey	{ VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.MPKCPublicKey}	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Er
_G	{ 0: 11010110111111111111011100110111 10010000010010001011111001100010 10001001101010111C	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Er
base	{ 0: 11010110111111111111011100110111 10010000010010001011111001100010 10001001101010111C	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Er
_length	14	int
_matrix	{ int[1608][ ] }	int[ ]
Length	14	int
_isDisposed	false	bool
_N	2048	int
_T	40	int
G	{ 0: 11010110111111111111011100110111 10010000010010001011111001100010 10001001101010111C	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Er
base	{ 0: 11010110111111111111011100110111 10010000010010001011111001100010 10001001101010111C	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Er
_length	14	int
_matrix	{ int[1608][ ] }	int[ ]
Length	14	int
K	1608	int
N	2048	int
Name	"MPKCPublicKey"	string
T	40	int
Static members		
ALG_NAME	"MPKCPublicKey"	string
privKey	null	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.In

Figure 34: Public Key (2)

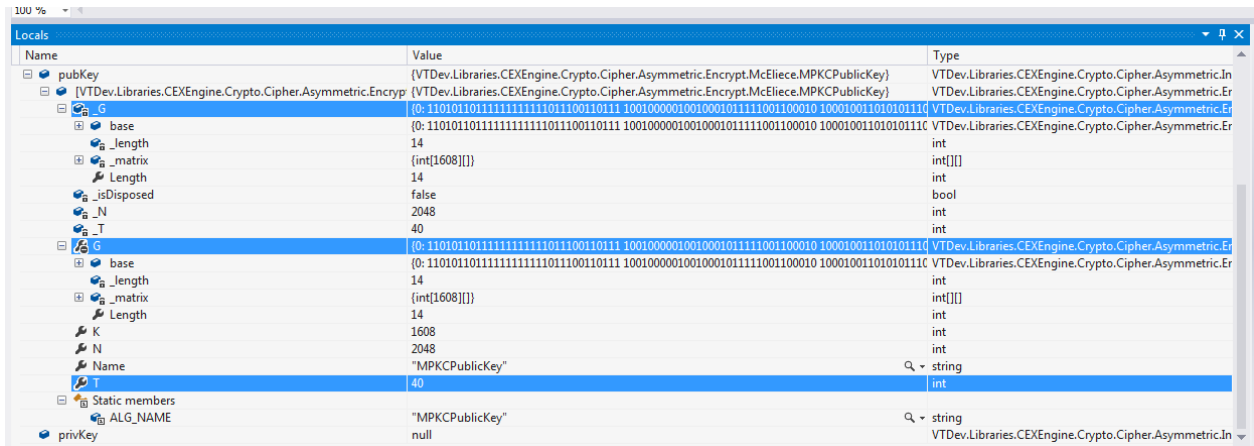


Figure 35: Public Key (3)

Length of public key is 88456 Bytes as shown in fig 36-37

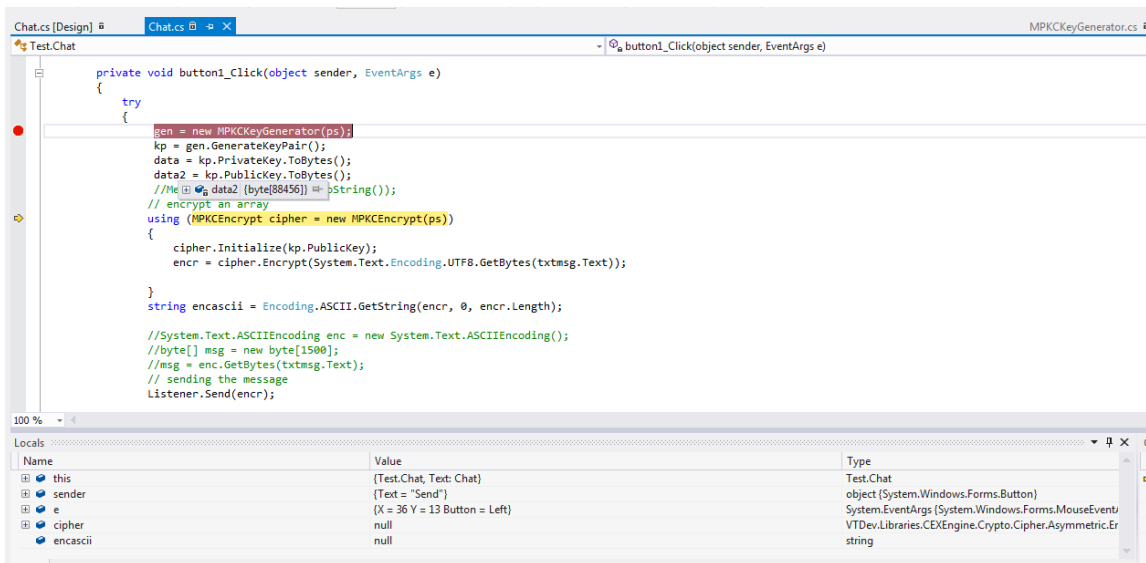


Figure 36: Length of Public Key (1)

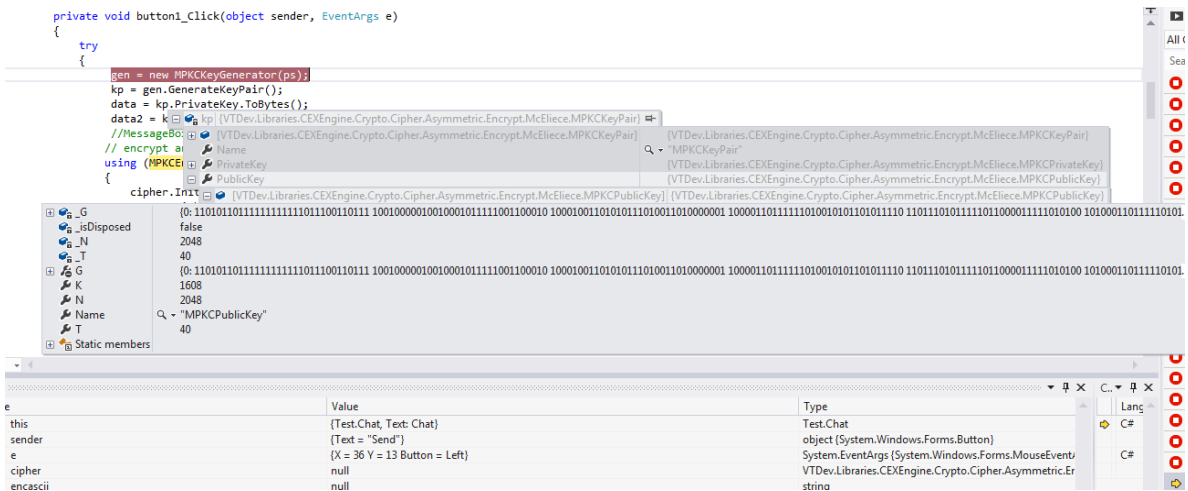


Figure 37: Public Key Generated (2)

**Generation of Private Key:** Fol parameters and matrices (which have already been defined/ generated previously ) are used to generate private key. Results are shown below

- "N">Length of the code
- "K">The dimension of the code
- "Gf">The finite field  $GF(2^m)$
- "Gp">The irreducible Goppa polynomial
- "P">The permutation matrix
- "H">The canonical check matrix
- "QInv">The matrix used to compute square roots in  $(GF(2^m))^t$

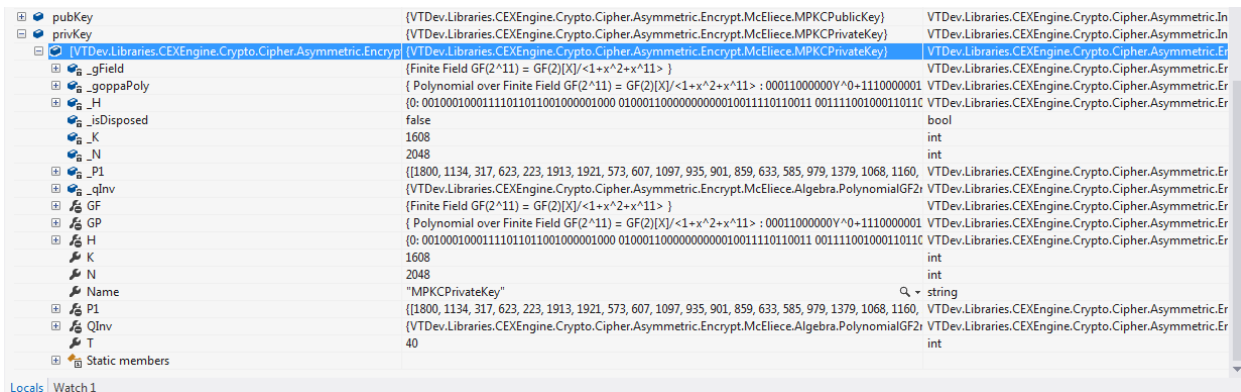


Figure 38: Private Key

Name	Value	Type
pubKey	[VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.MPKCPublicKey]	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.In
privKey	[VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Encrypt.McEliece.MPKCPrivateKey]	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Pr
gField	[Finite Field GF(2^11) = GF(2)[X]/<1+x^2+x^11>]	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Er
goppaPoly	{ Polynomial over Finite Field GF(2^11) = GF(2)[X]/<1+x^2+x^11> : 00011000000Y^0+1110000000	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Er
coefficients	[int[4]]	int[]
degree	40	int
field	[Finite Field GF(2^11) = GF(2)[X]/<1+x^2+x^11>]	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Er
degree	11	int
polynomial	2053	int
Degree	11	int
Polynomial	2053	int
Degree	40	int
Field	[Finite Field GF(2^11) = GF(2)[X]/<1+x^2+x^11>]	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Er
degree	11	int
polynomial	2053	int
Degree	11	int
Polynomial	2053	int
Head	1	int
Static members		

Figure 39: Private Key with fields

Length of Private Key is 119038 Bytes as shown in fig 40

```

private void button1_Click(object sender, EventArgs e)
{
    try
    {
        gen = new MPKCKeyGenerator(ps);
        kp = gen.GenerateKeyPair();
        data = kp.PrivateKey.ToBytes();
        data[0] = data[0] + (byte)(119038);
        //MessageBox.Show(data[2].ToString());
        // encrypt an array
        using (MPKCEncrypt cipher = new MPKCEncrypt(ps))
        {
            cipher.Initialize(kp.PublicKey);
            encr = cipher.Encrypt(System.Text.Encoding.UTF8.GetBytes(txtmsg.Text));
        }
        string encascii = Encoding.ASCII.GetString(encr, 0, encr.Length);

        //System.Text.ASCIIEncoding enc = new System.Text.ASCIIEncoding();
        //byte[] msg = new byte[1500];
        //msg = enc.GetBytes(txtmsg.Text);
        // sending the message
        Listener.Send(encr);
    }
}

```

Name	Value	Type
this	{Test.Chat, Text: Chat}	Test.Chat
sender	{Text = "Send"}	object (System.Windows.Forms.Button)
e	{X = 36 Y = 13 Button = Left}	System.EventArgs (System.Windows.Forms.MouseEvent)
cipher	null	VTDev.Libraries.CEXEngine.Crypto.Cipher.Asymmetric.Er

Figure 40: Length of Private Key

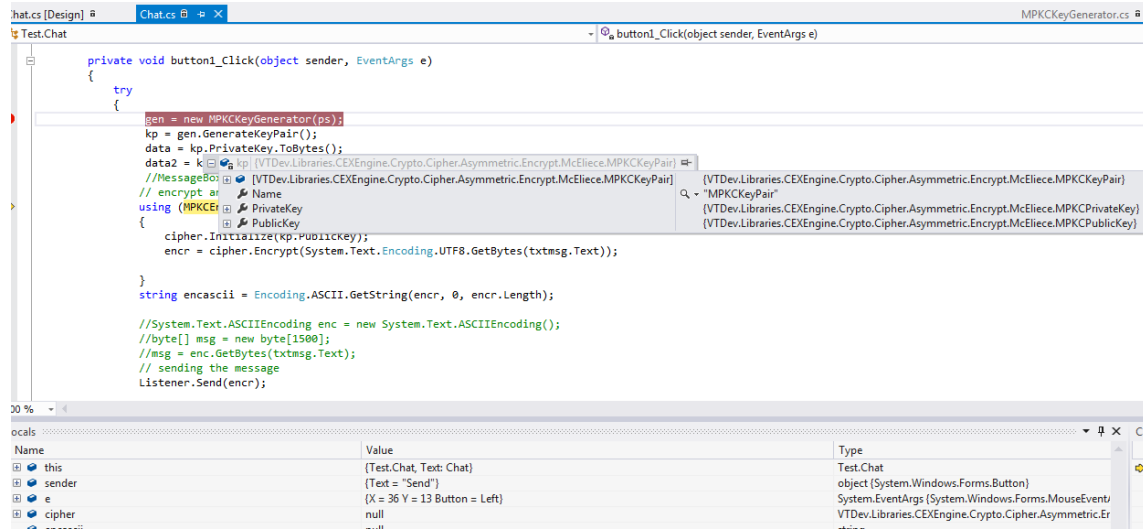


Figure 41: Private Key with parameters

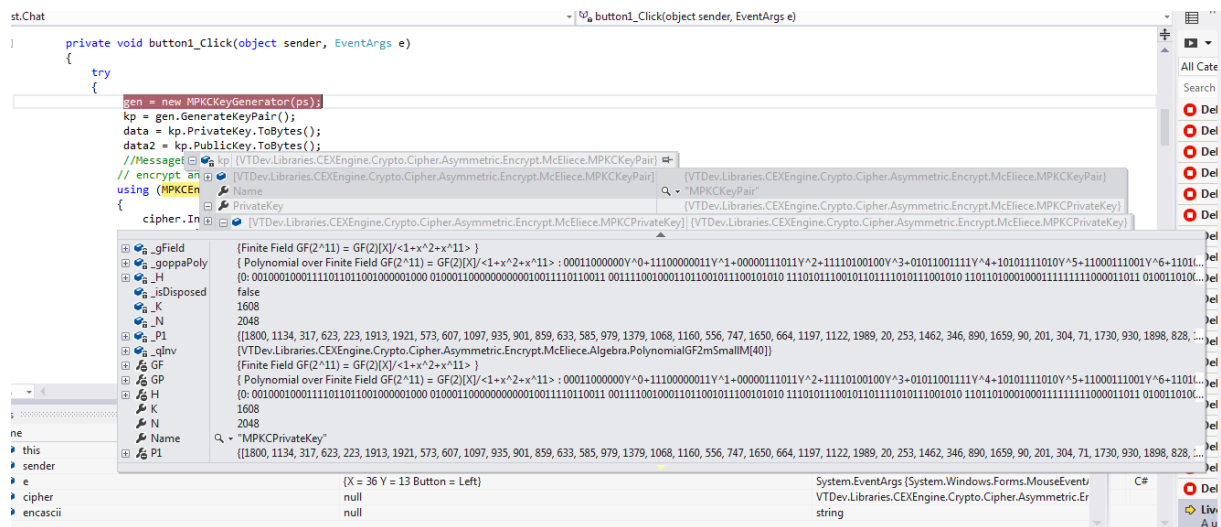


Figure 42: Generated Private Key

## 5.4 Encryption and Decryption Process

Next phase is encrypting the data by using public key and sending it to recipient. At receiving end encrypted data / cypher text is decrypted by using private key as shown in fig 43.

```

}

private void button1_Click(object sender, EventArgs e)
{
    try
    {
        gen = new MPKCKeyGenerator(ps);
        kp = gen.GenerateKeyPair();
        data = kp.PrivateKey.ToBytes();
        data2 = kp.PublicKey.ToBytes();
        //MessageBox.Show(data[2].ToString());
        // encrypt an array
        using (MPKCEncrypt cipher = new MPKCEncrypt(ps))
        {
            cipher.Initialize(kp.PublicKey);
            encr = cipher.Encrypt(System.Text.Encoding.UTF8.GetBytes(txtmsg.Text));
        }
        string encascii = Encoding.ASCII.GetString(encr, 0, encr.Length);

        //System.Text.ASCIIEncoding enc = new System.Text.ASCIIEncoding();
        //byte[] msc = new byte[15000];
    }
}

```

Figure 43: Encrypting Plain Text with Public Key

In the above fig 43 two variables are being declared as data and data2. Data type variable contain the Private Key while Data2 variable contain the generated Public Key.

The size of the Private key generated is 119038 Bytes whereas of Public Key is 88456 Bytes as shown in fig 44-46.

### 5.3.1 Private Key

```

{
    gen = new MPKCKeyGenerator(ps);
    kp = gen.GenerateKeyPair();
    data = kp.PrivateKey.ToBytes();
    dat = data (byte[119038]) -> s();
    //MessageBox.Show(data[2].ToString());
    // encrypt an array
    using (MPKCEncrypt cipher = new MPKCEncrypt(ps))
    {
        cipher.Initialize(kp.PublicKey);
        encr = cipher.Encrypt(System.Text.Encoding.UTF8.GetBytes(txtmsg.Text));
    }
    string encascii = Encoding.ASCII.GetString(encr, 0, encr.Length);
    //System.Text.ASCIIEncoding enc = new System.Text.ASCIIEncoding();
}

```

Figure 44: Separating Public and Private Keys



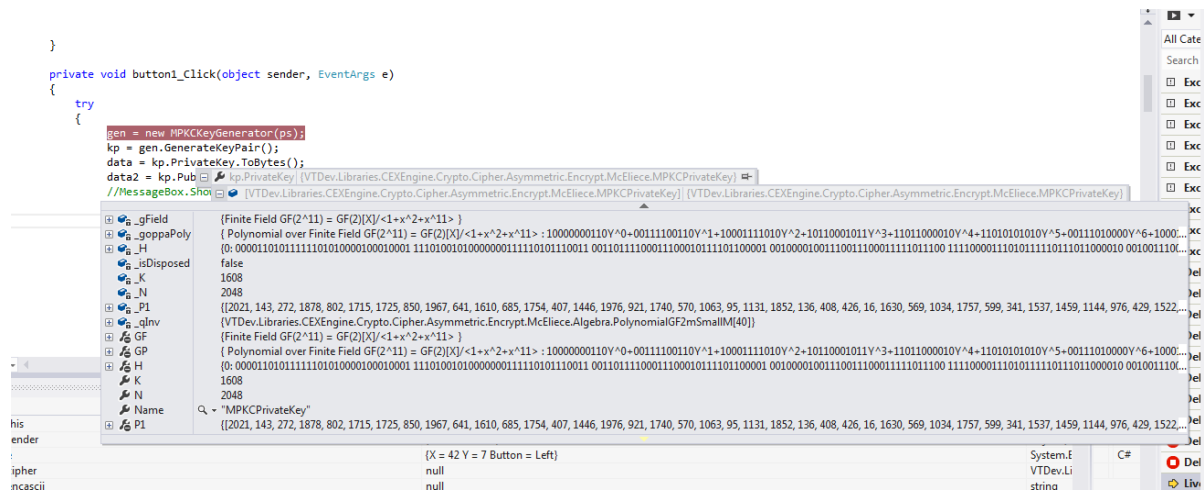


Figure 45: Private Key

### 5.3.2 Public Key

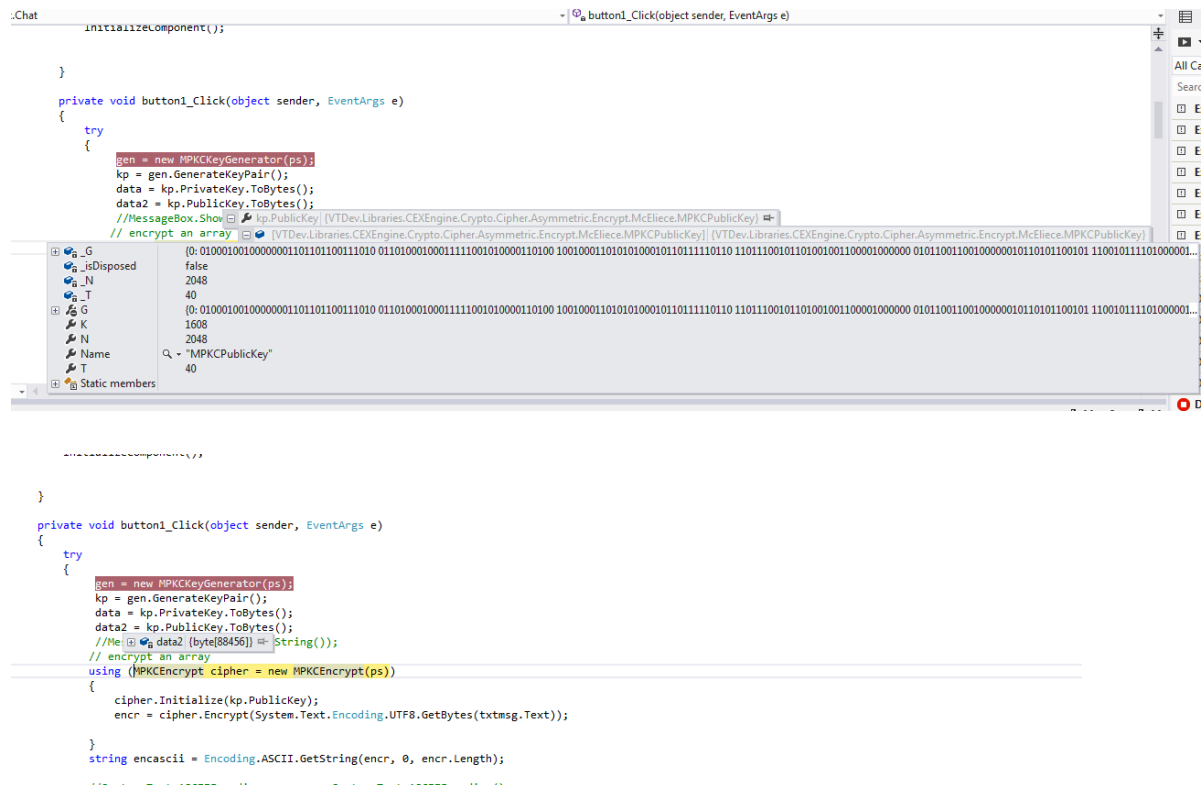


Figure 46: Public Key

In app interface plain text is entered in message field and corresponding cypher text is displayed in chat field as shown in fig 47. In this example we have added the text as “**Hello World**”.

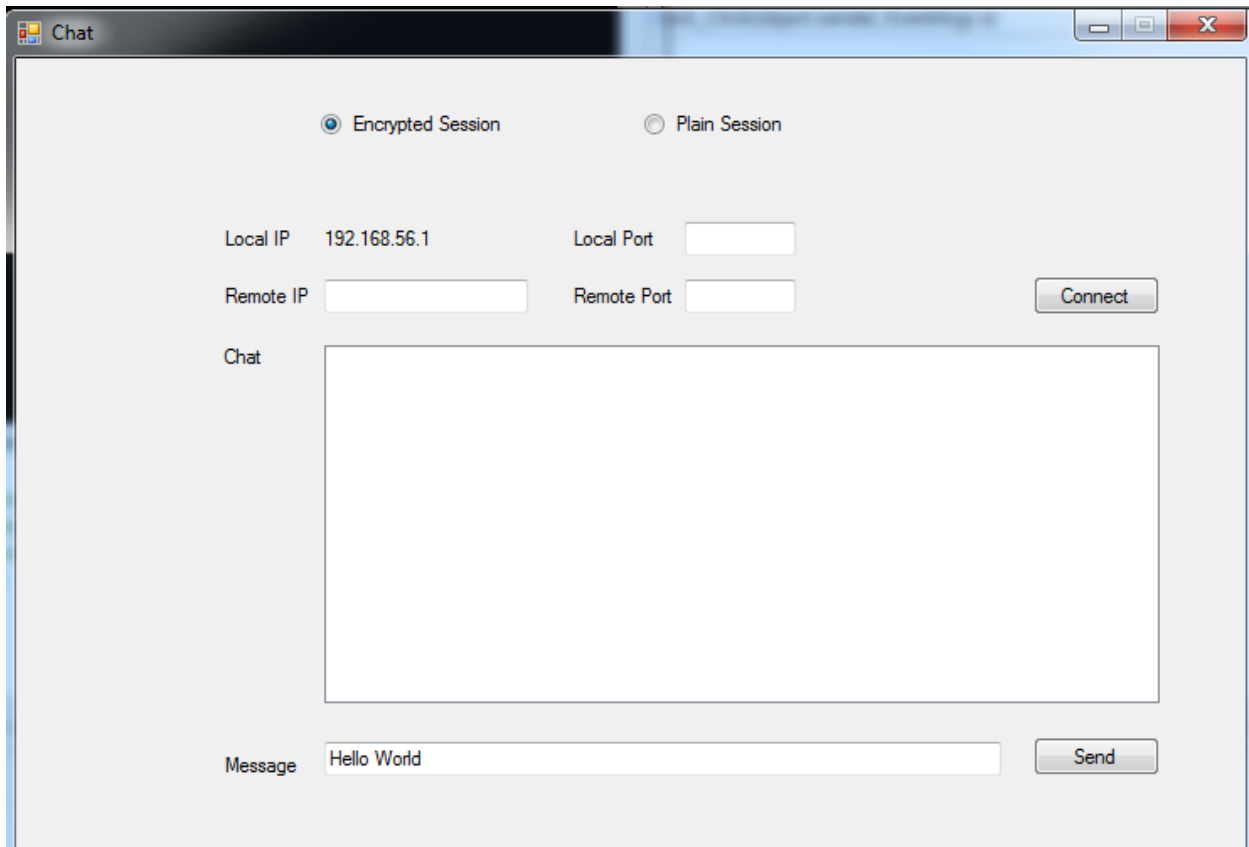


Figure 47: Sending Plain Text "Hello World"

The same plain text will be available in the plain text box in code that will be encrypted with the generated keys.

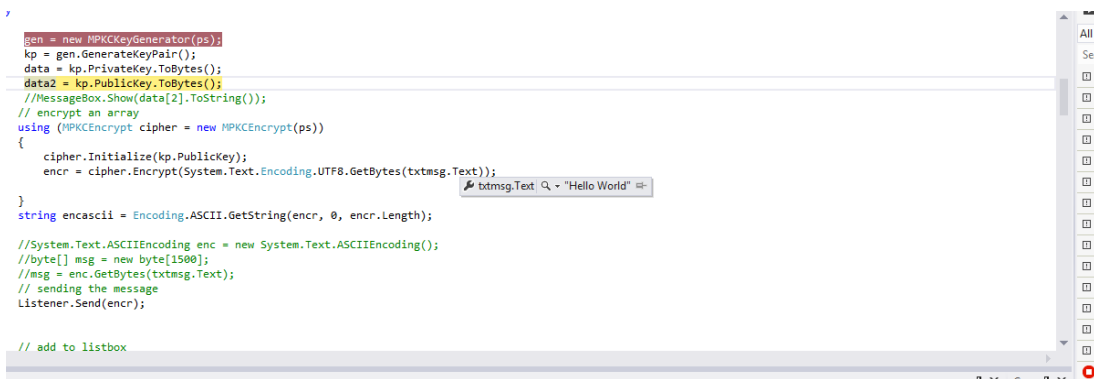


Figure 48: Plain Text in Debugging Mode

In fig 48 we can see that the text “Hello World” that we have entered is available and will be encrypted with the Public key. The resultant Encrypted Text will be stored in data variable “encr”.

Initially the encr type variable is initialized as Null but after encryption it will be populated with the cipher text as shown in fig 49.

```
try
{
    gen = new MPKKeyGenerator(ps);
    kp = gen.GenerateKeyPair();
    data = kp.PrivateKey.ToBytes();
    data2 = kp.PublicKey.ToBytes();
    //MessageBox.Show(data[2].ToString());
    // encrypt an array
    using (MPKCEncrypt cipher = new MPKCEncrypt(ps))
    {
        cipher.Initialize(kp.PublicKey);
        encr = cipher.Encrypt(System.Text.Encoding.UTF8.GetBytes(txtmsg.Text));
    }
    string encascii = Encoding.ASCII.GetString(encr, 0, encr.Length);

    //System.Text.AsciiEncoding enc = new System.Text.AsciiEncoding();
    //byte[] msg = new byte[1500];
    //msg = enc.GetBytes(txtmsg.Text);
    // sending the message
    Listener.Send(encr);

    // add to listbox
}
```

Figure 49: Encrypting plain text

```
try
{
    gen = new MPKKeyGenerator(ps);
    kp = gen.GenerateKeyPair();
    data = kp.PrivateKey.ToBytes();
    data2 = kp.PublicKey.ToBytes();
    //MessageBox.Show(data[2].ToString());
    // encrypt an array
    using (MPKCEncrypt cipher = new MPKCEncrypt(ps))
    {
        cipher.Initialize(kp.PublicKey);
        encr = cipher.Encrypt(System.Text.Encoding.UTF8.GetBytes(txtmsg.Text));
    }
    string encascii = Encoding.ASCII.GetString(encr, 0, encr.Length);

    //System.Text.AsciiEncoding enc = new System.Text.AsciiEncoding();
    //byte[] msg = new byte[1500];
    //msg = enc.GetBytes(txtmsg.Text);
    // sending the message
    Listener.Send(encr);
}
```

Figure 50: Bytes generated of plain Text

```

gen = new MPKKeyGenerator(ps);
kp = gen.GenerateKeyPair();
data = kp.PrivateKey.ToBytes();
data2 = kp.PublicKey.ToBytes();
//MessageBox.Show(data[2].ToString());
// encrypt an array
using (MPKCEncrypt cipher = new MPKCEncrypt(ps))
{
    cipher.Initialize(kp.PublicKey);
    encr = cipher.Encrypt(System.Text.Encoding.UTF8.GetBytes(txtmsg.Text));
}
string encr = Encoding.ASCII.GetString(encr, 0, encr.Length);
//System.Text.ASCIIEncoding enc = new System.Text.ASCIIEncoding();
//byte[] msg = new byte[1500];
//msg = enc.GetBytes(txtmsg.Text);
// sending the message
Listener.Send(encr);

// add to listbox
ChatArea.Items.Add("You: " + txtmsg.Text);
ChatArea.Items.Add("Cipher: " + encascii);

// clear txtMessage
txtmsg.Clear();
}
catch (Exception ex)

```

Figure 51: Encrypted Text generated

In fig 49-51 we can see that the encr type of variable now have 267 Bytes which is encrypted version of the entered Plain text.

This encrypted text is then sent to the receiver using the socket address as shown in fig 52

```

string encascii = Encoding.ASCII.GetString(encr, 0, encr.Length);

//System.Text.ASCIIEncoding enc = new System.Text.ASCIIEncoding();
//byte[] msg = new byte[1500];
//msg = enc.GetBytes(txtmsg.Text);
// sending the message
Listener.Send(encr);

// add to listbox
ChatArea.Items.Add("You: " + txtmsg.Text);
ChatArea.Items.Add("Cipher: " + encascii);

// clear txtMessage
txtmsg.Clear();
}
catch (Exception ex)

```

Figure 52: Sending Encrypted text to Recipient over socket

At receiver side this data is decrypted by using private key as shown in fig 53:-

```

int size = Listener.EndReceiveFrom(aResult, ref epRemote);
// check if theres actually information
if (size > 0)
{
    // used to help us on getting the data
    byte[] receivedData = new byte[1500];
    // getting the message data
    receivedData = (byte[])aResult.AsyncState;
    receivedData = TrimEnd(receivedData);
    // converts message data byte array to string
    ASCIIEncoding eEncoding = new ASCIIEncoding();
    string receivedMessage = eEncoding.GetString(receivedData);
    // decrypt the cipher text
    using (MPKCEncrypt cipher = new MPKCEncrypt(ps))
    {
        cipher.Initialize(kp.PrivateKey);
        dec = cipher.Decrypt(receivedData);
    }

    string asciiString = Encoding.ASCII.GetString(dec, 0, dec.Length);

    // adding Message to the listbox
    this.Invoke(new MethodInvoker(delegate()

```

Figure 53: Receiving Cipher text

The Variable type “dec” contains the decrypted data or plain text as received after decryption using the private key.

Numbers of Bytes as recovered after decryption are as shown in fig 54

```

// CHECK AT SENDER SECURITY INFORMATION
if (size > 0)
{
    // used to help us on getting the data
    byte[] receivedData = new byte[1500];
    // getting the message data
    receivedData = (byte[])aResult.AsyncState;
    receivedData = TrimEnd(receivedData);
    // converts message data byte array to string
    ASCIIEncoding eEncoding = new ASCIIEncoding();
    string receivedMessage = eEncoding.GetString(receivedData);
    // decrypt the cipher text
    using (MPKCEncrypt cipher = new MPKCEncrypt(ps))
    {
        cipher.Initialize(kp.PrivateKey);
        dec = cipher.Decrypt(receivedData);
    }
    string asciiString = Encoding.ASCII.GetString(dec, 0, dec.Length);

    // adding Message to the listbox
    this.Invoke(new MethodInvoker(delegate()

```

Figure 54: Received Cipher Text Bytes same as sent

At receiving end app interface this decrypted text is displayed in fig 55

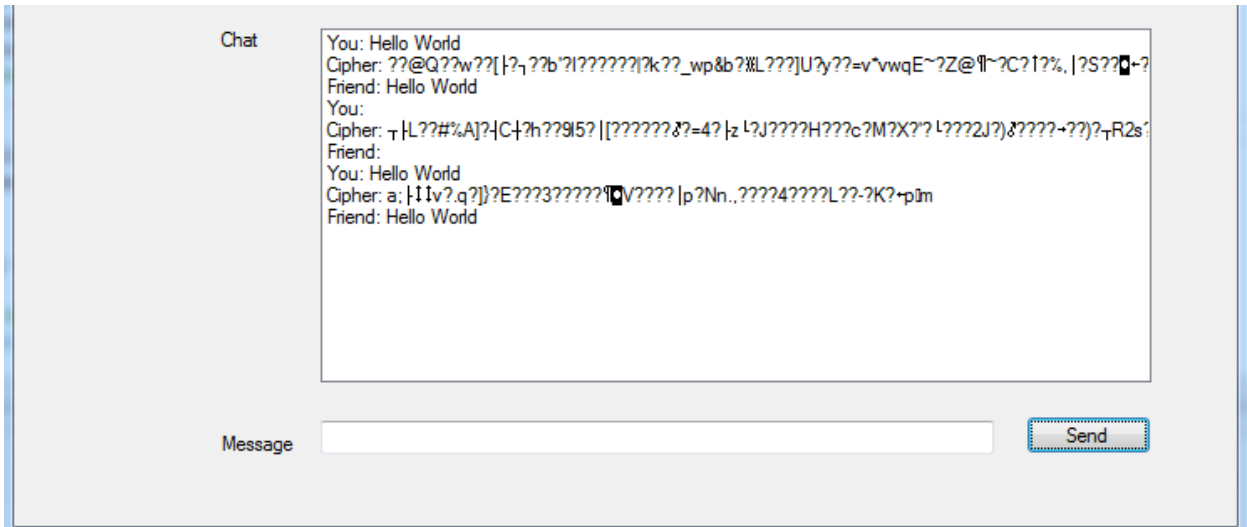


Figure 55: Sent and Received Cipher Text on GUI

## Chapter 6

### Discussion and Analysis

Before analysis on the results below are the specifications of the machine that we have used while implementing McEliece cryptosystem bases messaging application:-

- 13<sup>th</sup> Generation Core i7 13700HX
- 16 GB DDR5 RAM
- Windows 11
- 1 TB ROM SSD
- HM770 Chip Set
- Nvidia Graphics Card 8 GB GDDR6

The results that we have drawn in previous chapter we can summarize them as below:-

- Creation of parameters for the purpose of key generation
- Public and Private Key Generation
- Encryption Process of the plain text using McEliece Crypto system
- Decryption process of the received cipher text

We will do analysis of each portion one by one.

## **6.1 Creation of Parameters**

McEliece crypto system use different matrices along with polynomials for the purpose of key generation. These matrices are of different sizes and some are of random values. Below are listed the different Parameters generated along with their size that are used for the key generation process.

### **6.1.1 Finite Field:**

A finite field, also called Galois field, is a mathematical structure that consists of a finite set of elements along with two binary operations, addition and multiplication. It provides the necessary mathematical framework for the construction, encoding, and decoding of goppa codes used in the McEliece cryptosystem. It is also used in key generation, encryption and decryption in McEliececryptosystem. The use of finite fields ensures the security and reliability of the encryption and decryption processes which is required for the safety of sensitive information against attacks.

Table 4: Finite Field Analysis

Sr. No	Finite Field / Polynomial	Degree	Finite Field Generated	Remarks
1	Finite field $GF(2^m)$	11	$GF(2^{11}) = GF(2)[X]/\langle 1+x^2+x^{11} \rangle$	Finite fields can be constructed using different methods, such as polynomial basis representation, normal basis representation, or

				irreducible polynomial representation. We have used irreducible polynomial representation to achieve better security. The security of the system relies on the difficulty of decoding the code, which depend upon the properties of the irreducible polynomial.
--	--	--	--	---

**6.1.2 Goppa Polynomial:**

A Goppa polynomial is an irreducible polynomial with coefficients in a finite field which is explained above. It is used in construction of goppa codes, encoding and decoding process in mcEliece encryption. The selection of Goppa polynomial play crucial role in code length, code's error-correction capability, and resistance to various attacks, such as the Information Set Decoding (ISD) attack. So, properties/ attributes of Goppa polynomials are carefully considered to ensure the security of system against potential attacks and to achieve the desired performance goals in the encryption and decryption processes of the McEliece cryptosystem. Goppa polynomial we used in our app implementation is given below in table 5.

**Table 5: Goppa Polynomial Analysis**

Sr. No	Finite Field / Polynomial	Degree	Goppa Polynomial Over Finite Field
2.	Goppa Polynomial	40	$11101000011Y^0+11111010111Y^1+00100100101Y^2+0001101011Y^3+00111001111Y^4+01101100000Y^5+10011001110Y^6+00001100100Y^7+01110000101Y^8+0001100011Y^9+11111001101Y^{10}+11010111101Y^{11}+00110110011Y^{12}+11111110011Y^{13}+11000011110Y^{14}+01000100010Y^{15}+00010111000Y^{16}+00010000000Y^{17}+10010100100Y^{18}+01010111000Y^{19}+01110111111Y^{20}+010$



			$ \begin{aligned} &00001011Y^{21}+11110110000Y^{22}+10101001111Y^{23}+11 \\ &101000100Y^{24}+11111000001Y^{25}+01011001100Y^{26}+1 \\ &1100000001Y^{27}+10110101101Y^{28}+00100101010Y^{29}+ \\ &00000011010Y^{30}+01101001001Y^{31}+10010110000Y^{32} \\ &+10001000110Y^{33}+10001000001Y^{34}+00000001000Y^{35} \\ &+10000011101Y^{36}+01000010111Y^{37}+11100111000Y^{38} \\ &+10101101010Y^{39}+00000000001Y^{40};\} \end{aligned} $
--	--	--	--

Next are the different matrices that we have generated for the purpose of generating the Public and Private keys. The analysis part of these is listed below in table 6:

**Table 6: Generated Matrices Analysis**

Sr. No	Matrix Name	Purpose	Size
1.	qInv	It is used for computing square roots in $(GF(2^m))^t$	Coefficients 40, Degree 39
2.	h (canonical check matrix)	It is a compact or systematic form of a parity check matrix which is in binary form. It is used in decoding algorithm to identify and correct errors in received codewords. Syndrome (used to determine the error pattern and apply error correction to the received codeword) of the received word is obtained by multiplying it with canonical check matrix.	64
3.	p (Permutation Matrix)	Permutation matrix is a square matrix with 1 in each row and each column, and remaining entries being zero. It is part of private key which is used in encryption and decryption processes of mcEilece cryptosystem. It adds an extra layer of security by introducing	2048

		randomness and complexity, which makes the cryptosystem resistant to various potential attacks.	
4.	s (Singular Matrix)	It is non invertible matrix which is used in encryption and decryption process of mcEliece cryptosystem as shown in previous chapter. During encryption process it is used to multiply the encoded message and inverse of the same matrix is used during decryption process. It add and additional layer of security in mcEliece cryptosystem against attacks.	16
4.	G ( Generator Matrix)	Generator matrix is a binary matrix which is the basic component used in encryption and decryption processes of mcEliececryptosytem. It is part of both public and private keys. It is generated using a specific form of the error-correcting code, in our case by using Goppa codes. The construction involves selecting appropriate code parameters which includes code length, dimension, and minimum distance, to achieve the required error-correction capability, properties and security level for the cryptosystem.	
4.	Short G	It is systematic form of generator matrix. Systematic form of generator matrix makes the encoding and decoding process of mcEliece cryptosystem simple and more efficient which makes it suitable for implementing the encryption and decryption operations in the	16

		McEliece cryptosystem and reduce computational complexity	
--	--	---	--

## 6.2 Number of Random Vectors tested while decryption

Table 7: Random Variable testing

Parameters (n,k,m,t)	Number of Random Vector Tested
( 2048, 1707, 11, 31)	7000
( 4096, 3604, 12, 41)	20500
( 4096, 3352, 12, 62)	11000
( 6960, 5413, 13, 119)	250

## 6.3 Initial Parameters vs. Key length

As we have already explained in the theoretical part of the algorithm that the length of the public and private key is dependent upon the initial values of the size of the matrix. So, we have tested the application with the different values of the k and n to determine the exact size of the Public and Private Key. The results thus obtained are as shown in table 7: -

Table 8: Key Size VS. Initial Parameters

Sr. No	K	N	Max Text Length	Public Key Size	Private Key Size	Security Level
1.	1608	2048	201	88488	119071	Low
2.	1520	2048	190	100368	142531	Low
3.	3724	4096	465	175076	200119	Low to Medium
4.	3604	4096	450	223496	262519	Medium
5.	3520	2096	440	253488	306371	Medium
6.	3448	4096	431	306371	344039	Medium

7.	3292	4096	431	332540	425928	High
8.	7815	8192	976	375168	403733	High
9.	7620	8192	952	548688	604893	High

From the results we have obtained above we can say that by increasing the values of the K and N the size of the Public and Private Key increase which ultimately results in increase in security level of the application.

#### **6.4 Encryption and Decryption Time**

In order to analyze the application, we have also inserted the timer in the application with the help of which we can calculate the encryption and decryption time of the plain text. The results thus obtained are as shown in table 8: -

Table 9: Encryption and Decryption Time with plain text length

Sr. No	Text Entered	Encryption Time (ms)	Decryption Time (ms)
1.	A	404	46
2.	Aa	950	31
3.	Aaa	648	31
4.	Aaaa	308	46
5.	Aaaaa	886	31
6.	Aaaaaa	918	46
7.	Aaaaaaa	404	46
8.	Aaaaaaaa	591	46
9.	Aaaaaaaaa	497	31
10.	Aaaaaaaaaa	528	31

## **Conclusion and Future Work**

In this thesis we have successfully implemented the McEliece Cryptosystem based secure chat application. Security of most of communication systems/ apps is furnished by public cryptographic systems. Presently used public cryptosystems i.e. RSA and ECC are considered secure in the presence of conventional computers. Underlying hard problems of these systems are factorization and discrete logarithm problems which are unable to solve by conventional computers. However, Quantum computers which are in their early stage of development will someday be able to break the security systems now in use because they can efficiently solve these problems that conventional computers cannot. In order to be ready for the day when quantum computers are a practical reality, it is required to look at additional computing issues

that are equally challenging for both conventional and quantum computers. In this regard we short listed McEliece Crypto System which is code based cryptographic algorithm and is quantum resistive as well. We chose this scheme/ cryptosystem because it is one of the few cryptosystems that is expected to withstand assaults from powerful computers in the future, despite several efforts by the crypto community to break it, it is secure till today. Before developing the application, we did a complete in-depth analysis of this crypto system and literature review on work that has already been done in this field. The application has been developed using the Visual studio software and using C Shrap language. The final product that we will be providing to the two users is just an exe file which will be installed in the computer and shortcut as well as the icon will be visible on the desktop. All the required repositories and data files along with the encryption algorithms will be placed in installation folder automatically. Once the socket established between the two users, they can easily communicate with each other. It may be used between any two clients who are using an unsecured channel or media. Nowadays, the majority of applications employ traditional encryption methods; which will be insecure in the presence of quantum computers. The security of the developed application is based on McEliece cryptosystem. As the underlying cryptosystem of this app is McEliece cryptosystem which is a quantum resistive cryptosystem so this app is also secure against quantum attacks in other words we can say that it is a quantum resistive messaging application. Moreover, this application may be integrated with financial systems to facilitate secure transactions. The secure nature of app can prevent unauthorized access to sensitive financial information, making it harder for hackers to intercept or manipulate transactions. This can enhance the security and integrity of digital payments, banking, and other financial services. It can also be used in Government and military organizations for communication. Governments and military organizations deal with highly sensitive information, and secure communication is crucial for their operations. This app can ensure that their messages remain confidential and resistant to interception by adversaries.

As far as the future work is concerned, we have used the socket programming to connect two clients who are intended to communicate each other using our application which means that a dedicated IP connectivity and free socket will be required for establishment of connection before communication start. Due to this limitation, we can use this application in private LAN or private

dedicated WAN environment only. However, by embedding port translation function it can be used on the internet as well.

Another limitation of the application is that it is computer-based application where it utilizes the processing power and other resources of the CPU to calculate the crypto key pairs. Some algorithms can be modified to run this application on mobile phone / android or any other platform.

Moreover, we have designed this Quantum resistive application for chat purpose only, this can be enhanced for images, videos and files as well.

## **References**

[1] Ghosh, Santosh, et al. "A speed area optimized embedded co-processor for McEliece cryptosystem." *2012 IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors*. IEEE, 2012.

[2] Chikouche, Nouredine, et al. "RFID authentication protocols based on error-correcting codes: a survey." *Wireless Personal Communications* 96 (2017): 509-527.

[3] Shor, Peter W. "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer." *SIAM review* 41.2 (1999): 303-332.

[4] Perlner, Ray A., and David A. Cooper. "Quantum resistant public key cryptography: a survey." *Proceedings of the 8th Symposium on Identity and Trust on the Internet*. 2009.

- [5] Diffie, Whitfield, and Martin E. Hellman. "New directions in cryptography." *Democratizing Cryptography: The Work of Whitfield Diffie and Martin Hellman*. 2022. 365-390.
- [6] Minder, Lorenz, and Amin Shokrollahi. "Cryptanalysis of the Sidelnikov cryptosystem." *Advances in Cryptology-EUROCRYPT 2007: 26th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Barcelona, Spain, May 20-24, 2007. Proceedings 26*. Springer Berlin Heidelberg, 2007.
- [7] Manin, Y. "Computable and incomputable. Sovetskoye Radio, 1980."
- [8] Feynman, Richard P. "Simulating physics with computers." *Feynman and computation*. CRC Press, 2018. 133-153.
- [9] McEliece, Robert J. "A public-key cryptosystem based on algebraic." *Coding Thv 4244* (1978): 114-116.
- [10] Niederreiter, Harald. "Knapsack-type cryptosystems and algebraic coding theory." *Prob. Contr. Inform. Theory 15.2* (1986): 157-166.
- [11] Sidelnikov, Vladimir Michilovich, and Sergey O. Shestakov. "On insecurity of cryptosystems based on generalized Reed-Solomon codes." (1992): 439-444.
- [12] Sidelnikov, Vladimir Michilovich. "A public-key cryptosystem based on binary Reed-Muller codes." (1994): 191-208.
- [13] Kandasamy, Ilanthenral, and K. S. Easwarakumar. "Chained Hexi Codes Signature Scheme." *International Journal of Computer Science and Network Security (IJCSNS)* 14.12 (2014): 20.
- [14] Greenwell, Raymond N. "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer." *The College Mathematics Journal* 31.1 (2000): 70.
- [15] Baldi, Marco. *QC-LDPC code-based cryptography*. Springer Science & Business, 2014.
- [16] Berger, Thierry P., and Pierre Loidreau. "How to mask the structure of codes for a cryptographic use." *Designs, Codes and Cryptography* 35 (2005): 63-79.



- [17] Baldi, Marco, and Franco Chiaraluce. "Cryptanalysis of a new instance of McEliece cryptosystem based on QC-LDPC codes." *2007 IEEE International Symposium on Information Theory*. IEEE, 2007.
- [18] Wieschebrink, Christian. "Cryptanalysis of the Niederreiter public key scheme based on GRS subcodes." *Post-Quantum Cryptography: Third International Workshop, PQCrypto 2010, Darmstadt, Germany, May 25-28, 2010. Proceedings 3*. Springer Berlin Heidelberg, 2010.
- [19] Janwa, Heeralal, and Oscar Moreno. "McEliece public key cryptosystems using algebraic-geometric codes." *Designs, Codes and Cryptography* 8.3 (1996): 293-307.
- [20] Loidreau, Pierre. "Strengthening McEliece cryptosystem." *ASIACRYPT*. 2000.
- [21] Shrestha, Sujan Raj, and Young-Sik Kim. "New McEliece cryptosystem based on polar codes as a candidate for post-quantum cryptography." *2014 14th International Symposium on Communications and Information Technologies (ISCIT)*. IEEE, 2014.
- [22] Chabaud, Florent. "On the security of some cryptosystems based on error-correcting codes." *Workshop on the Theory and Application of Cryptographic Techniques*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994.
- [23] Xinmei, Wang. "Digital signature scheme based on error-correcting codes." *Electronics Letters* 26.13 (1990): 898-899.
- [24] Faugere, Jean-Charles, et al. "A distinguisher for high-rate McEliece cryptosystems." *IEEE Transactions on Information Theory* 59.10 (2013): 6830-6844.
- [25] Hall, Jonathan I. *Notes on coding theory*. FreeTechBooks. com, 2003.
- [26] Sendrier, Nicolas. "On the concatenated structure of a linear code." *Applicable Algebra in Engineering, Communication and Computing* 9.3 (1998): 221-242.
- [27] Niederreiter, Harald. "Knapsack-type cryptosystems and algebraic coding theory." *Prob. Contr. Inform. Theory* 15.2 (1986): 157-166.
- [28] Sidelnikov, Vladimir Michilovich, and Sergey O. Shestakov. "On insecurity of cryptosystems based on generalized Reed-Solomon codes." (1992): 439-444.

- [29] Sidelnikov, Vladimir Michilovich. "A public-key cryptosystem based on binary Reed-Muller codes." (1994): 191-208.
- [30] Oswald, Elisabeth, and Marc Fischlin, eds. *Advances in Cryptology—EUROCRYPT 2015: 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II*. Vol. 9057. Springer, 2015.
- [31] Shooshtari, MasumehKoochak, Mahmoud Ahmadian, and Ali Payandeh. "Improving the security of McEliece-like public key cryptosystem based on LDPC codes." *2009 11th International Conference on Advanced Communication Technology*. Vol. 2. IEEE, 2009.
- [32] Faure, Cédric, and Lorenz Minder. "Cryptanalysis of the McEliece cryptosystem over hyperelliptic codes." *Proceedings of the 11th international workshop on Algebraic and Combinatorial Coding Theory, ACCT*. Vol. 2008. 2008.
- [33] Couvreur, Alain, Irene Márquez-Corbella, and Ruud Pellikaan. "A polynomial time attack against algebraic geometry code based public key cryptosystems." *2014 IEEE International Symposium on Information Theory*. IEEE, 2014.
- [34] Monico, Chris, Joachim Rosenthal, and Amin Shokrollahi. "Using low density parity check codes in the McEliece cryptosystem." *2000 IEEE International Symposium on Information Theory (Cat. No. 00CH37060)*. IEEE, 2000.
- [35] Garelo, R., et al. "Quasi-Cyclic Low-Density Parity-Check Codes in the McEliece Cryptosystem." *Titolo volume non avvalorato*. 2007.
- [36] Löndahl, Carl, and Thomas Johansson. "A new version of McEliece PKC based on convolutional codes." *Information and Communications Security: 14th International Conference, ICICS 2012, Hong Kong, China, October 29-31, 2012. Proceedings 14*. Springer Berlin Heidelberg, 2012.
- [37] Landais, Grégory, and Jean-Pierre Tillich. "An efficient attack of a McEliece cryptosystem variant based on convolutional codes." *Post-Quantum Cryptography: 5th International Workshop, PQCrypto 2013, Limoges, France, June 4-7, 2013. Proceedings 5*. Springer Berlin Heidelberg, 2013.

- [38] Li, Yuan Xing, Robert H. Deng, and Xin Mei Wang. "On the equivalence of McEliece's and Niederreiter's public-key cryptosystems." *IEEE Transactions on Information Theory* 40.1 (1994): 271-273.
- [39] Gabidulin, Ernst M., A. V. Paramonov, and O. V. Tretjakov. "Ideals over a non-commutative ring and their application in cryptology." *Advances in Cryptology—EUROCRYPT'91: Workshop on the Theory and Application of Cryptographic Techniques Brighton, UK, April 8–11, 1991 Proceedings 10*. Springer Berlin Heidelberg, 1991.
- [40] Gabidulin, Ernst M., et al. "Reducible rank codes and their applications to cryptography." *IEEE Transactions on Information Theory* 49.12 (2003): 3289-3293.
- [41] Gaborit, Philippe. "Shorter keys for code based cryptography." *Proceedings of the 2005 International Workshop on Coding and Cryptography (WCC 2005)*. 2005.
- [42] Sidelnikov, Vladimir Michilovich. "A public-key cryptosystem based on binary Reed-Muller codes." (1994): 191-208.
- [43] Janwa, Heeralal, and Oscar Moreno. "McEliece public key cryptosystems using algebraic-geometric codes." *Designs, Codes and Cryptography* 8.3 (1996): 293-307.
- [44] Overbeck, Raphael. "A new structural attack for GPT and variants." *Progress in Cryptology—Mycrypt 2005: First International Conference on Cryptology in Malaysia, Kuala Lumpur, Malaysia, September 28-30, 2005. Proceedings 1*. Springer Berlin Heidelberg, 2005.
- [45] Kobara, Kazukuni, and Hideki Imai. "On the one-wayness against chosen-plaintext attacks of the Loidreau's modified McEliece PKC." *IEEE Transactions on Information Theory* 49.12 (2003): 3160-3168.
- [46] Kobara, Kazukuni, and Hideki Imai. "Semantically secure McEliece public-key cryptosystems-conversions for McEliece PKC." *Public Key Cryptography: 4th International Workshop on Practice and Theory in Public Key Cryptosystems, PKC 2001 Cheju Island, Korea, February 13–15, 2001 Proceedings 4*. Springer Berlin Heidelberg, 2001.
- [47] Sendrier, Nicolas. "Code-based cryptography: State of the art and perspectives." *IEEE Security & Privacy* 15.4 (2017): 44-50.
- [48] Xinmei, Wang. "Digital signature scheme based on error-correcting codes." *Electronics Letters* 26.13 (1990): 898-899..

- [49] Harn, L., and D. C. Wang. "Cryptanalysis and modification of digital signature scheme based on error-correcting code." *Electronics Letters* 2.28 (1992): 157-159.
- [50] Alabbadi, Mohssen, and Stephen B. Wicker. "A digital signature scheme based on linear error-correcting block codes." *International Conference on the Theory and Application of Cryptology*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994.
- [51] Stern, Jacques. "Can one design a signature scheme based on error-correcting codes?." *ASIACRYPT*. Vol. 94. 1994.
- [52] Courtois, Nicolas T., Matthieu Finiasz, and Nicolas Sendrier. "How to achieve a McEliece-based digital signature scheme." *Advances in Cryptology—ASIACRYPT 2001: 7th International Conference on the Theory and Application of Cryptology and Information Security Gold Coast, Australia, December 9–13, 2001 Proceedings* 7. Springer Berlin Heidelberg, 2001.
- [53] Stern, Jacques. "A new identification scheme based on syndrome decoding." *Annual International Cryptology Conference*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993.
- [54] Augot, Daniel, Matthieu Finiasz, and Nicolas Sendrier. "A family of fast syndrome based cryptographic hash functions." *Mycrypt*. Vol. 3715. 2005.
- [55] Fischer, Jean-Bernard, and Jacques Stern. "An efficient pseudo-random generator provably as secure as syndrome decoding." *Advances in Cryptology—EUROCRYPT'96: International Conference on the Theory and Application of Cryptographic Techniques Saragossa, Spain, May 12–16, 1996 Proceedings* 15. Springer Berlin Heidelberg, 1996.
- [56] Shor, Peter W. "Algorithms for quantum computation: discrete logarithms and factoring." *Proceedings 35th annual symposium on foundations of computer science*. Ieee, 1994.
- [57] Daniel J. Bernstein (2009). "Introduction to post-quantum cryptography" (PDF). Post-Quantum Cryptography.
- [58] Gershon, Eric. "New qubit control bodes well for future of quantum computing." *Phys.org* (2014).
- [59] Monica, Heger. "Cryptographers take on quantum computers." *Cryptography* 2 (2018): 1.

- [60] Buchmann, Johannes, and Jintai Ding, eds. *Post-Quantum Cryptography: Second International Workshop, PQCrypto 2008 Cincinnati, OH, USA October 17-19, 2008 Proceedings*. Vol. 5299. Springer Science & Business Media, 2008.
- [61] Mosca, Michele. "Setting the scene for the etsi quantum-safe cryptography workshop." *e-proceedings of 1st Quantum-Safe-Crypto Workshop, Sophia Antipolis*. 2013.
- [62] Bernstein, Daniel J. "Cost analysis of hash collisions: Will quantum computers make SHARCS obsolete." *SHARCS 9* (2009): 105.
- [63] Bernstein, Daniel J. "Grover vs. mceliece." *Post-Quantum Cryptography: Third International Workshop, PQCrypto 2010, Darmstadt, Germany, May 25-28, 2010. Proceedings 3*. Springer Berlin Heidelberg, 2010.
- [64] Peikert, Chris. "Lattice cryptography for the internet." *Post-Quantum Cryptography: 6th International Workshop, PQCrypto 2014, Waterloo, ON, Canada, October 1-3, 2014. Proceedings 6*. Springer International Publishing, 2014.
- [65] Güneysu, Tim, Vadim Lyubashevsky, and Thomas Pöppelmann. "Practical lattice-based cryptography: A signature scheme for embedded systems." *Cryptographic Hardware and Embedded Systems—CHES 2012: 14th International Workshop, Leuven, Belgium, September 9-12, 2012. Proceedings 14*. Springer Berlin Heidelberg, 2012.
- [66] Zhang, Jiang, et al. "Authenticated key exchange from ideal lattices." *Advances in Cryptology-EUROCRYPT 2015: 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part II 34*. Springer Berlin Heidelberg, 2015.
- [67] Ducas, Léo, et al. "Lattice signatures and bimodal Gaussians." *Advances in Cryptology—CRYPTO 2013: 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*. Springer Berlin Heidelberg, 2013.
- [68] Lyubashevsky, Vadim, Chris Peikert, and Oded Regev. "On ideal lattices and learning with errors over rings." *Journal of the ACM (JACM)* 60.6 (2013): 1-35.

- [69] Daniel, Augot, and B. Lejla. "Initial recommendations of long-term secure post-quantum systems." *PQCRYPTO. EU. Horizon 2020* (2015).
- [70] Stehle, D., and R. Steinfeld. "Making NTRUencrypt and NTRUSign as secure as standard worst-case problems over ideal lattices." *Cryptology ePrint Archive, Report 2013/004* (2013).
- [71] Easttom, Chuck. "An analysis of leading lattice-based asymmetric cryptographic primitives." *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*. IEEE, 2019.
- [72] Ding, Jintai, and Dieter Schmidt. "Rainbow, a new multivariable polynomial signature scheme." *ACNS*. Vol. 5. 2005.
- [73] Buchmann, Johannes, Erik Dahmen, and Andreas Hülsing. "XMSS-a practical forward secure signature scheme based on minimal security assumptions." *Post-Quantum Cryptography: 4th International Workshop, PQCrypto 2011, Taipei, Taiwan, November 29–December 2, 2011. Proceedings 4*. Springer Berlin Heidelberg, 2011.
- [74] Bernstein, Daniel J., et al. "The SPHINCS+ signature framework." *Proceedings of the 2019 ACM SIGSAC conference on computer and communications security*. 2019.
- [75] Huelsing, A., et al. "RFC 8391: XMSS: eXtended Merkle Signature Scheme." (2018).
- [76] Naor, Moni, and Moti Yung. "Universal one-way hash functions and their cryptographic applications." *Proceedings of the twenty-first annual ACM symposium on Theory of computing*. 1989.
- [77] Overbeck, Raphael, and Nicolas Sendrier. "Code-based cryptography." *Post-quantum cryptography* (2009): 95-145.
- [78] Jao, David, and Luca De Feo. "Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies." *Post-Quantum Cryptography: 4th International Workshop, PQCrypto 2011, Taipei, Taiwan, November 29–December 2, 2011. Proceedings 4*. Springer Berlin Heidelberg, 2011.

- [79] Higgins, Parker. "Pushing for perfect forward secrecy, an important web privacy protection." (2013).
- [80] Carracedo, Jorge Martiez, et al. "Cryptography for security in IoT." *2018 Fifth International Conference on Internet of Things: Systems, Management and Security*. IEEE, 2018.
- [81] Perlner, Ray; Cooper (2009). "Quantum Resistant Public Key Cryptography: A Survey". NIST. Retrieved 23 Apr 2015.
- [82] Campagna, Matt, et al. "Kerberos revisited quantum-safe authentication." *ETSI Quantum-Safe-Crypto Workshop*. 2013.
- [83] Lyubashevsky, Vadim, Chris Peikert, and Oded Regev. "On ideal lattices and learning with errors over rings." *Journal of the ACM (JACM)* 60.6 (2013): 1-35.
- [84] Akleylek, Sedat, et al. "An efficient lattice-based signature scheme with provably secure instantiation." *Progress in Cryptology–AFRICACRYPT 2016: 8th International Conference on Cryptology in Africa, Fes, Morocco, April 13-15, 2016, Proceedings 8*. Springer International Publishing, 2016.
- [85] Nejatollahi, Hamid, et al. "Post-quantum lattice-based cryptography implementations: A survey." *ACM Computing Surveys (CSUR)* 51.6 (2019): 1-41.
- [86] Bulygin, Stanislav, Albrecht Petzoldt, and Johannes Buchmann. "Towards provable security of the unbalanced oil and vinegar signature scheme under direct attacks." *Progress in Cryptology-INDOCRYPT 2010: 11th International Conference on Cryptology in India, Hyderabad, India, December 12-15, 2010. Proceedings 11*. Springer Berlin Heidelberg, 2010.
- [87] Pereira, Geovandro CCF, Cassius Puodzius, and Paulo SLM Barreto. "Shorter hash-based signatures." *Journal of Systems and Software* 116 (2016): 95-100.
- [88] Garcia, LC Coronado. "On the security and the efficiency of the Merkle signature scheme." *Technical Report 2005/192, Cryptology ePrint Archive* (2005).
- [89] McEliece, Robert J. *Information, coding and mathematics*. Springer Science & Business Media, 2002.

- [90] Wang, Yongge. "Quantum resistant random linear code based public key encryption scheme RLCE." *2016 IEEE International Symposium on Information Theory (ISIT)*. IEEE, 2016.
- [91] Grimes, Roger A. *Cryptography apocalypse: preparing for the day when quantum computing breaks today's crypto*. John Wiley & Sons, 2019.
- [92] Delfs, Christina; Galbraith (2013). "Computing isogenies between supersingular elliptic curves over  $F_p$ ". arXiv:1310.7789 [math.NT].
- [93] Hirschhorn, Philip S., et al. "Choosing NTRUEncrypt parameters in light of combined lattice reduction and MITM approaches." *Applied Cryptography and Network Security: 7th International Conference, ACNS 2009, Paris-Rocquencourt, France, June 2-5, 2009. Proceedings 7*. Springer Berlin Heidelberg, 2009.
- [94] Petzoldt, Albrecht, Stanislav Bulygin, and Johannes Buchmann. "Selecting parameters for the rainbow signature scheme." *Post-Quantum Cryptography: Third International Workshop, PQCrypto 2010, Darmstadt, Germany, May 25-28, 2010. Proceedings 3*. Springer Berlin Heidelberg, 2010.
- [95] Bürstinghaus-Steinbach, Kevin, et al. "Post-quantum tls on embedded systems: Integrating and evaluating kyber and sphincs+ with mbedtls." *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. 2020.
- [96] Chopra, Arjun. "Glyph: A new instantiation of the glp digital signature scheme." *Cryptology ePrint Archive* (2017).