An implementation oriented overview of code based Cryptography

A thesis submitted towards partial fulfillment of
5 year integrated BS-MS programme
by
Kapil Kumar Gupta
Under the guidance of
Dr. Bimal K. Roy
(Director, ISI kolkata)

Department of Mathematics
Indian Institute of Science Education and Research Pune

Certificate

This is to certify that this dissertation entitled " **An implementation oriented overview of code based cryptography** " towards the partial fulfillment of the 5 year integrated BS-MS programme at the Indian Institute of Science Education and Research Pune, represents original research carried out by Kapil Kumar Gupta at Cryptography Research Group, ISI- Kolkata, under the supervision of Dr. Bimal K. Roy during the academic year 2010-2011.

Name and Signature of the Candidate

Supervisor: Dr. Bimal K Roy
(Director, ISI kolkata)
Date: 8-April-2011
Place: ISI, Kolkata

Head: Mathematical Science
Date: 8-April-2011
Place: Pune

Contents

Acknowledgement

This thesis would have been impossible without support of many people who were always there to help me. I gratefully acknowledge aid from the following wonderful sources.
At first I would like to thank Dr. Nithin Nagaraj who suggested me the name of such a big personality in the field of cryptography for my thesis work. He is the first person, who created a great interest of mine in the field of cryptography. Even though he was a visiting faculty at my institute but he was also a great advisor of mine at every moment.
I would like to give my special thanks to Dr. Soumen Maity for recommending me to my thesis advisor Dr. Bimal Roy. He boosted my knowledge in the field of public key cryptography. I would also like to thank you for being my internal thesis advisor. I hope this thesis work will be able to complete your expectation from me as being my advisor.
I would like to thank Prof. Bimal K. Roy(Director, ISI kolkata) for accepting me under his supervision for my masters thesis work. It was great experience being at ISI kolkata campus, and working in your group. I would also like to thank you for introducing me to Dr. Bhaskar Biswas, (visiting scientist at ISI kolkata). Dr. Bhaskar helped me a lot in my project even though having a busy schedule of him. He suggested me a great topic for my thesis work. You are great personality both as advisor and as friend.
I would also like to thank Dr. Harald Niederreiter , who helped me through mail in my research work.
Last but not least, I would like to thank: Rahul, Debaditya, Neha, Sharmishtha, Kunwarbir,Inam, Snehashish, Radhika, Aditya, Jeet, Harneesh, Souvik, Sourish, Devasis, Aritra, Subhajeet, Tarun, Krishna and many other friends for supporting at every point of time both in project and visiting the city.
Once again I would like to say thanks to all the people for supporting me during my masters thesis and also would like to say sorry to those people, whose name may be I may have forgotten to mention.

Abstract

Currently Quantum computer are expected to be biggest threat in public key cryptography. My thesis work is about study and practical implementation of the code based cryptosystem. This thesis is inspired by the research work done by Dr. Bhaskar Biswas during his ph.d. at INRIA, Paris. Here we have discussed the improvised version of Mc'Eliece Cryptosystem. Improvised version is known as HYMES(Hybrid McEliece Encription Scheme). Thesis is divided in three parts: Theory, Implementation and Further Research work. Theory parts cover the mathematical background behind this cryptosystem and also study of the original code based crypto-schemes. Implementation of the project is done in the "C" programming language, this part includes study of necessary pseudo codes. We have also tried to make some new additions in Niederreiter Cryptosystem, these improvements will help to make encryption and decryption faster than before. Hence we have given a brief detail about these additions in Further Research Work.

Introduction

Here we start with a brief introduction of Public Key Cryptography and Coding Theory. For better understanding of coding theory we will also introduce basics of Group Theory, Field Theory and Galois Theory. Later we will move to Public Key Cryptography based on theory of error correction-detection code, emphasizing on McEliece Cryptosystem and Niederriter Cryptosystem.

Public Key Cryptography

The data transferred from one system to another over public network can be protected by the method of encryption. In traditional cryptography, the sender and receiver of a message know and use the same secret key; the sender uses the secret key to encrypt the message, and the receiver uses the same secret key to decrypt the message. This type of cryptosystem is called the Symmetric key cryptosystem. Here main challenge is to distribute the key between sender and receiver so that no one else come to know about the key. In this case both sender and receiver have to agree for the key.

In order to solve the key management problem, Whitfield Diffie and Martin Hellman introduced the concept of public-key cryptography in 1976 [6]. Public-key cryptography is a cryptographic approach which involves the use of asymmetric key algorithms instead of or in addition to symmetric key algorithms. This cryptosystem doesn't require initial key exchange problem or we some time call it key-management problem. Security of this cryptosystem depends on the difficulties of prime factorization, logarithmic problems and other algebraic problems.

Key generation is the most important part of the public key cryptosystem. Key generation algorithm generates the two pairs of keys, one is used as a public key and other is used as private key and it is computationally hard to calculate the private key from the given pairs of public key.

Any public key cryptosystem consists of following three components

1. *Key generation scheme*: This scheme generates pair of keys using the difficulties of algebra and number theory. One pair is published for encryption by the sender which is called public key and the second pair is used for decryption, called private key.

2. *Encryption function*: The encryption function $\mathbb{E}(m, K_{public})$, takes the message m (as an integer) and public key to compute corresponding cipher text c.

3. *Decryption function*: The decryption function $\mathbb{D}(c, K_{private})$, takes the cipher text c (as an integer) and private key to regenerate corresponding plain text m.

Some Public Key Cryptography schemes are listed below:

1. Diffie−Hellman key exchange protocol
2. ElGamal
3. RSA encryption algorithm
4. Merkle−Hellman knapsack cryptoshceme
5. DSS (Digital Signature Standard), which incorporates the Digital Signature Algorithm
6. Various elliptic curve techniques
7. McEliece Cryptosystem
8. Niederriter Cryptosystem

There are many other PKC schemes but they are not widely used.

PKC based on coding theory

Currently known quantum computers are nowhere near powerful enough to attack real crypto systems, many cryptographers are researching new algorithms, in case quantum computing becomes a threat in the future. Hence the cryptosystem based on the coding theory play an important role. It has a firm, well developed mathematical background. This is also considered as Post-quantum cryptography scheme (Schemes which are unbreakable using quantum computer also in polynomial time using **Shor's algorithm**[18]). In contrast, most current symmetric cryptography

(symmetric ciphers and hash functions) is secure from quantum computers. Thus post-quantum cryptography does not focus on symmetric algorithms.
Most of the previous work have been devoted to cryptanalysis and semantic security, but very few attempt have been made to examine implementation issues. Implementing public key cryptosystem creates a compromise between security and efficiency, this means to get the more security we need the large parameter hence less efficiency. For that reason, cryptanalysis and implementation have to be considered in unison.
Here we are going to discuss McEliece cryptosystem first. There have been many modification to this cryptosystem and Niederreiter is one of its variant.
Regarding security, code based cryptosystems are more safer than other public key cryptosystem. There many attacks have been made on McEliece cryptosystem but most of them are decoding attacks.
Here we have divided our thesis in three parts. First part contains the *Theory* behind cryptosystem used and second part contains the details of implementation.
In theory Chapter 1 contains the algebraic background needed for the implementation. This chapter is divided in different sections which contains the details of different field of algebra and basic algorithms. In second chapter we will discuss about coding theory. This chapter will contain some basic definitions and concepts of coding theory. Third chapter will contain the details of McEliece and Niederrieter cryptosystem. In this chapter we had present The Hybrid McEliece Encription Scheme($HYMES$). This is an improved version of original McEliece cryptosystem. Idea of HYMES was given by Biswas and Sendrier [4]. This new hybrid cryptosystem has following two modifications:

- Increase the information rate by putting some data in error pattern
- Reduce the public key size by using reducible row-echelon form of the generator matrix.

Biswas used "Constant weight encoding "[17] method to encode the information into error pattern. Here in my thesis we are not including this step. In my thesis work I am generating the random error pattern as given in original cryptosystem. Implementation of his work is given on the following weblink:
http://www-rocq.inria.fr/secret/CBCrypto/index.php?pg=hymes
The Idea of McEliece system is to hide the structure of code by means of transformation of generator matrix. The transformed generator matrix becomes the public key and the trapdoor information is the structure of goppa code together with the transformation parameters. The security relies on the fact that the decoding problem for general linear code is NP-complete.

Part I

Theory

Chapter 1

Algebraic Background

This chapter will cover the theoretical background, needed for the implementation of code based cryptography scheme. Here we need a brief introduction of field theory and Galois theory. We will start with definition of *finite field*, its basic properties, then will move to field extension, its different representations and algorithms used for the implementation. I have also briefly described basics of coding theory, the error correcting codes and binary goppa codes with details, in next chapter.

1.1 Finite Fields

A *field* $(\mathbb{F},\ +,\ *)$ is a set of elements, with two binary operations, "+"and "$*$", such that it satisfies the following axioms.

1. $\mathbb{F}$ is a additive group(Closed under addition and there exist an identity 0_F, and additive inverse '$-a$ 'for all $a \in \mathbb{F}$).
2. $\mathbb{F} - \{0\}$ is a multiplicative group(Closed under multiplication and there exist identity '1_F'and a multiplicative inverse, 'b', for all $a \in \mathbb{F}$, such that $a * b = 1$, here b is unique for every a) .
3. It follows the distributive property. $a * (b + c) = a * b + a * c$ holds for all $a,\ b,\ c \in \mathbb{F}$,

A field with finite number of elements is called *finite field*. Some important definitions are as follows

Definition 1.1.1 *(Characteristic):Characteristic of a field, is a smallest integer p, s.t. $p1_F = 0$, where 1_F is multiplicative identity of the field. If no such p exist, the characteristic is assumed to be 0.*

Definition 1.1.2 *(Prime Fields): For any prime p, $\mathbb{F}_p = \mathbb{Z}/\mathbb{Z}_p$, is called prime field.*

Definition 1.1.3 *(Order): Order of field is the number of element in the field. For any p prime a field always has the order equals to p^m, for some integer m.*

Definition 1.1.4 *(Subfield): Any subset, of a field $\mathbb{F}$, which satisfies the field axioms,with same operations, is called subfield.*

Characteristic of field is either 0 or prime p. Field genetated by multiplicative identity is called prime-subfield, which is isomorphic to $\mathbb{F}_p$ (characteristic p) or $\mathbb{Q}$ (characteristic 0).

1.2 Field Extension

Later we are going to use this section extensively, hence we have to study this section thoroughly.

Definition 1.2.1 *If K is a field containing the subfield F, then K is said to be an extension field, or extension of field F, denoted as K/F.*

If K is a field with no subfield, then it is called prime field.

Definition 1.2.2 *Primitive element of a finite field F is a generator of the multiplicative group of the field, which is necessarily cyclic.*

Definition 1.2.3 *The degree of field extension K/F is the demension of K as a vector space over the field F. It is denoted by $[K : F]$. Hence $[K : F]$=$dim_F(K)$*

Theorem 1.2.4 *If F is a finite field of cardinality $q \geq 2$, then*

1. *$q = p^n$, where p is some prime and n is any integer.*
2. *$\mathbb{F}$ is unique upto isomorphism.*

Definition 1.2.5 *If the field K is generated by a single element α over F, $K = F(\alpha)$, then K is said to be a simple extension of F, and the element α is called primitive element for the extension.*

Definition 1.2.6 *Trace of any element $\alpha \in F_{p^m}$ over the field F_p is defined by*

$$Tr_{F_{p^m}/F}(\alpha) = \alpha + \alpha^p + ... + \alpha^{p^{m-1}}$$

If $K = F(\alpha)$, then $K \simeq \frac{F[x]}{p(x)}$, where $p(x) \in F[x]$ is an irreducible polynomial of degree $[K : F]$. Hence every element of field K can be written in the form of a polynomial of degree n, where n is degree of extension of field K over F.
More information about the primitive polynomial is given in the next section, which help us to understand the properties of polynomial.

1.3 Polynomials

This section is very important part for our implementation work. Our main interest is to disucss about the propertis of polynomials of finite degree.
Let R be a commutative ring with identity. The following expression

$$f(x) = \sum_{i=0}^{i=n} a_i x^i = a_0 + a_1 x + a_2 x^2 + ... + a_n x^n$$

where $n \geq 0$, each coefficient $a_i \in R$ and x is called indeterminate over R. If $n \neq 0$, then it is called degree of polynomial $f(x)$. Set of all polynomials over any ring R also forms a ring, which is denoted as $R[x]$. Properties of $R[x]$ are following

1. $R[x]$ is *commutative* if and only if R is *commutative*.
2. $R[x]$ have identity similar to the R.
3. $R[x]$ is an *integral domain* if and only if R is an *integral domain*.

If every element of $R[x]$ have an inverse in $R[x]$, then $R[x]$ is itself a field. This infact bring the notation of *irreducible polynomial*. We can define addition on the elements of $F_p[x]$. If $f(x) = \sum_{i=0}^{i=n} a_i x^i$ and $g(x) = \sum_{i=0}^{i=m} b_i x^i$ are two polynomial in $F_p[x]$, then addition can be defined as follows

$$f(x) + g(x) = \sum_{i=0}^{i=max(m,n)} (a_i + b_i)x^i)$$

Addition of a_i and b_i is done modulo p, because they both are elements of field F_p. Similarly we can also define the multiplication of two polynomial over $F_p[x]$.

$$f(x)g(x) = \sum_{k=0}^{k=m+n} (\sum_{i+j=k} (a_i b_j) x^k)$$

Definition 1.3.1 *(Irreducible Polynomial) A polynomial $p(x) \in F_p[x]$ is said to be irreducible over field F_p if this polynomial doesn't have any root in F_p or it can't be written as a product of two polynomial of $F_p[x]$, which are not unit.*

Definition 1.3.2 *A primitive polynomial is a polynomial that generates all elements of an extension field from a base field.*

Theorem 1.3.3 *For $p(x) \in F_p[x]$,the residue class ring $F_p[x]/p(x)$ is a field if and only if $p(x)$ is irreducible over F_p.*

1.4 Vector Space

A vector space is a mathematical structure formed by a collection of vectors.

Definition 1.4.1 *A vector space is a set V over a field $\mathbb{F}$ together with two binary operators('+'and '.') that satisfy axioms listed below. Elements of V are called vectors. Elements of F are called scalars. Then V is a vector if '+'and '.'operators satisfy the following axioms. If u, v, $w \in V$ and $a, b \in \mathbb{F}$*

- *(Group) V is a commutative group under addition('+').*
- *(Distributivity) $a.(v + w) = a.v + a.w$ and $(a + b).v = a.v + b.v$.*
- *(Associativity) $a.(b.v) = (a.b).v$*

$\mathbb{F}$ is called the scalar field of the vector V.

Definition 1.4.2 *A subset S of V is called a set of linearly independent vectors if an equation $\alpha_1 v_1 + \alpha_2 v_2 + ... + \alpha_n v_n$, with $\alpha_1, \alpha_2, ...\alpha_n \in \mathbb{F}$ and $v_1, v_2, ...v_n \in S$ implies $\alpha_1 = \alpha_2 = ... = \alpha_n = 0$. Otherwise they are called linearly dependent.*

Definition 1.4.3 *A basis of a vector space V is an ordered set of linearly independent vectors which spans V. In particular two bases will be considered different if one is simply an arrangement of the other. This is sometimes referred to as an ordered basis.*

Definition 1.4.4 *If $\phi : V \longrightarrow U$ is a linear transformation of vector spaces over $\mathbb{F}$, ker(ϕ) is sometimes called null space of ϕ, and the dimension of ker (ϕ) is called the nullity of ϕ. If ker $\phi = 0$, the transformation is said to be nonsingular.*

This theory part will help us a lot in implementing the code. We are going to work on extension field F_{p^m} over F_p, so using the representation of field elements will help us in implementing the code. Most commonly we used to represent the field element in the form of power of the primitive element. If α is a primitive element of F_{p^m}, then it can be written as

$$F_{p^m} = \{0, 1, \alpha, \alpha^2, ..., \alpha^{p^{m-2}}\}$$

Definition 1.4.5 *In the field F_{p^m}, there exits a β, such that*

$$\{\beta, \beta^p, ..., \beta^{p^{m-1}}\}$$

are linearly independent, where p is the characteristic of the field. Then this set forms a normal basis for F_{p^m}.

This basis is frequently used in cryptographic applications that are based on the discrete logarithm problem such as elliptic curve cryptography. Hardware implementations of normal basis arithmetic typically have far less power consumption than other bases.

Definition 1.4.6 *A natural representation of finite field elements as polynomial over a ground field is known as standard basis. It is also known as polynomial basis. Let $\alpha \in F_{p^m}$ be the root of a primitive polynomial of degree m over F_p. The polynomial basis of F_{p^m} is then*

$$\{1, \alpha, ..., \alpha^{m-1}\}$$

Thus every element of F_{p^m} can be represent in the form of polynomial of degree $m - 1$

$$a_0 + a_1\alpha + a_2\alpha^2 + ... + a_{m-1}\alpha^{m-1}$$

where $a_i \in F_p$. Every polynomial of above type can also be written as some power of α using *Zech's Logarirthm [11]*. Because of its simplicity standard basis representation has been widely used.
Similar to the linear algebra, concept of *dual basis* can be applied in context of finite

extensions. A dual basis isn't a concrete basis like the *standard basis* or the *normal basis*; rather it provides a way of using a second basis for computations. Consider two basis set B_1 and B_2 in F_{p^m}

$$B_1 = \alpha_0, \alpha_1, ..., \alpha_{m-1}$$

and

$$B_2 = \beta_0, \beta_1, ..., \beta_{m-1}$$

then B_2 can be consider as dual basis of B_1, provided

$$Tr(\alpha_i.\beta_j) = \{ \begin{array}{ll} 0, & if \quad i \neq j \\ 1, & otherwise \end{array}$$

Using a dual basis can provide a way to easily communicate between devices that use different bases, rather than having to explicitly convert between bases using the change of bases formula. Furthermore, if a dual basis is implemented then conversion from an element in the original basis to the dual basis can be accomplished with a multiplication by the multiplicative identity.

Example(F_{2^3}) F_{2^3} is an extension over F_2(binary field), where degree of extension is 3. Choose $p(x) = x^3 + x + 1$ as a primitive irreducible polynomial. Let α is a root of $p(x)$. So we can represent F_8 as

$$F_8 \cong \frac{F_2[x]}{x^3 + x + 1}$$

$F_2[x]$, is a ring of polynomials over F_2. Hence we can represent its element in order form and in polynomial form as given below

0	0
α^0	1
α^1	α
α^2	α^2
α^3	$1 + \alpha$
α^4	$\alpha + \alpha^2$
α^5	$1 + \alpha + \alpha^2$
α^6	$1 + \alpha^2$

Using the coefficient of the polynomial we can represent an element of the field F_8 as an m-tuple. Power reresentation helps us in mathematical framework, while $m - tuple$ representation helps in dealing with digital hardware.

1.5 Matrices

Idea of matrices comes in the picture for solving the system of linear equation. Moreover in linear algebra, a linear transformation of vector space can be completely determined by matrix. General vector spaces do not possess multiplication operation.

Definition 1.5.1 *(**Algebra:**) A vector space equipped with an additional bilinear operator, defining the multiplication of two vectors is called an algebra over a field* $\mathbb{F}$.

As we have already discussed in the section 1.4, that an extension field $\mathbb{F}_{2^m}$ can be treated as vector space with the ground field F_2 and whose dimension is m. We get a matrix representation by choosing a basis and taking its linear transformation with coefficients in the field $\mathbb{F}_{2^m}$.
For a given polynomial $p(x)$, A is its *companion matrix*[9] of size $(deg(p)+1) \times (deg(p)+1)$, we have $p(A) = 0$. The matrix A generates the cyclic group $< A >$ of order $m-1$, which is isomorphic to multiplicative field $\mathbb{F}^*_{2^m}$. Hence ring of matrices

$$\mathbb{F}_2[A] = \{0, I, A, A^2, ..., A^{m-2}\}$$

Which is also isomorphic to the extension field $\mathbb{F}_{2^m}$ over $\mathbb{F}$.

Definition 1.5.2 *(**Row-Echlon Form:**)In linear algebra a matrix is in row echelon form if*

- *All nonzero rows (rows with at least one nonzero element) are above any rows of all zeroes, and*
- *The leading coefficient (the first nonzero number from the left, also called the pivot) of a nonzero row is always strictly to the right of the leading coefficient of the row above it.*

We have following example of row-echelon form of matrix

$$\begin{pmatrix} 1 & a_1 & a_2 & a_3 \\ 0 & 1 & a_4 & a_5 \\ 0 & 0 & 1 & a_6 \end{pmatrix}$$

A matrix is in reduced row echelon form (also called row canonical form) if every leading coefficient is 1 and is the only nonzero entry in its column, like in this

example:

$$\begin{pmatrix} 1 & 0 & 0 & b_1 \\ 0 & 1 & 0 & b_2 \\ 0 & 0 & 1 & b_3 \end{pmatrix}$$

For implementing McEliece algorithm we need to have our generator matrix in reduced row-echelon form, for that detailed algorithm we are going to discuss in implementation chapter. We shall need to define the matrix multiplication as well. Let $\alpha \in \mathbb{F}_{2^m}$, be a field element, hence we will consider the following statements

- Binary vector representation of field element α, size of the vector will be m.
- Polynomial representation of α. If $\alpha = \alpha_0\alpha_1.....\alpha_{m-1}$,$(\alpha_i \in \{0,1\})$, then its polynomial representation $p(x) = \sum_{i=0}^{m-1} \alpha_i x^i$
- Matrix representation

$$A.g(x) = \begin{pmatrix} g(x) & \\ xg(x) & mod(p(x)) \\ \vdots & \vdots \\ x^{m-1}g(x) & mod(p(x)) \end{pmatrix}$$

  A is a non-singular binary matrix, where row shows the vector representation of α.

1.6 Algorithms:

Here we are discussing those algorithm which would later help us in implementation. Theoretical background needed for the algorithms we have already discussed in previous sections. Algorithms are as follows

1.6.1 Gaussian Elimination

In linear algebra, Gaussian elimination is an algorithm for solving systems of linear equations, finding the rank of a matrix, and calculating the inverse of an invertible square matrix. Elementary row operations are used to reduce a matrix to row echelon form. GaussJordan elimination, an extension of this algorithm, reduces the matrix further to reduced *row echelon form.*
Gaussian elimination writes a given $m \times n$ matrix A uniquely as a product of an

invertible $m \times m$ matrix S and a row-echelon matrix T. Here, S is the product of the matrices corresponding to the row operations performed. We will discuss about the pseudo code of this algorithm in chapter *implementation*.
Row Echelon form of a matrix is not unique. Every non-zero matrix can be reduced to many echelon forms using elementary matrix transformations. However, all matrices and their row echelon forms correspond to exactly one matrix in reduced row echelon form.
Gaussian elimination writes a given $m \times n$ matrix A uniquely as a product of an invertible $m \times m$ matrix S and a row-echelon matrix T. Here, S is the product of the matrices corresponding to the row operations performed.

1.6.2 Euclidean Algorithm

The Euclidean algorithm calculates the greatest common divisor (GCD) of two integers a and b. Similar algorithm is also used for GCD of two polynomials. If there are two polynomials $p, q \in F_{q^m}$, then a polynomial of maximum degree which can divide (without any remainder) both polynomial p and q is called greatest common divisor of polynomials p and q. In mathematics, the Euclidean algorithm [5] (also called Euclid's algorithm) is an efficient method for computing the greatest common divisor (GCD).
Let $f\&g$ are two polynomials in $F_{2^m}[x]$, then we can use division algorithm to write

$$f = p * g + r$$

where p and r are two polynomials in $F_{2^m}[x]$, such that $deg(r) \leq deg(g)$ or $r = 0$.

- If $r = 0$, this implies $f = g * p$, means f is divisible by g. Hence $GCD(f, g) = cg$, where $c \in F_{2^m}[x]$.
- If $r \neq 0$ then $GCD(f, g) = GCD(g, r)$ (can easily be checked using basics of number theory).

This step reduces the degree of polynomial involved, and so repeating the procedure leads to the greatest common divisor of two polynomials in the finite number of steps. GCD of g and f can be written as below:

$$GCD(g, f) = a.g + b.f$$

where a and b are two polynomials.
The extended Euclidean algorithm is an extension to the Euclidean algorithm for finding the greatest common divisor (GCD) of polynomials f and g: it also finds the polynomials a and b in F_{2^m}.

1.6.3 Irreducibility Check

To check a polynomial is irreducible or not, we have to find its factors. If a polynomial $p(x) \in F_{2^m}[x]$ can't be factorized in the field $F_{2^m}[x]$ then polynomial p is called irreducible in the field $F_{2^m}[x]$.
A given polynomial $p(x) \in F[x]$ has a factor of degree i if and only if, it has a common factor with $x^{2^{im}} - x$. Hence, we can say that if $\forall i \leq deg(p(x))/2$, GCD($p(x)$, $x^{2^{im}} - x$)=1, then $p(x)$ is an irreducible polynomial in the field F_{2^m}. The polynomial $x^{2^{im}} - x$, has very much high degree to handle directly. Hence in place of computing it directly we use an indirect method, where we calculate $h_j(x) = x^{2^j}$ mod(p(x)), by successive squaring modulo(p(x)). Hence:

$$GCD(p(x), x^{2^{im}} - x) = GCD(p(x), h_{im}(x) - x)$$

this step greatly simplifies the calculation.

1.6.4 Zech's logarithm

Zech logarithms are also called Jacobi Logarithm. Zech's logarithms are used with finite fields to reduce a high-degree polynomial that is not in the field to an element in the field (thus having a lower degree). Unlike the traditional logarithm, the Zech's logarithm of a polynomial provides an equivalence it does not alter the value.
Let $\alpha \in \mathbb{F}_{2^m}$ is a primitive root of the polynomial $f(x)$ over $\mathbb{F}_2$, then z(n) Zech's logarithm of integer n may be defined such that:

$$\alpha^{z(n)} = 1 + \alpha^n$$

This implies that $z(n) = log_\alpha(1 + \alpha^n)$, if $\alpha^n = -1$, then zech's logarithm is not defined. Zech's logarithms are also used when finite field elements are represented exponentially. The produce of α^a and α^b, will be $\alpha^a.\alpha^b = \alpha^{(a+b) \ mod \ 2^m-1}$. Here addition not as easy as multiplication, addition will be as following(assume that $b \geq a$):

$$\alpha^a + \alpha^b = \alpha^a(1 + \alpha^{b-a \ mod \ (2^m-1)}) = \alpha^{a+z(b-a)}$$

$z(b-a)$ can be obtained from the zech's log table [1]

Chapter 2

Coding Theory

Claude Shannons 1948 paper "A Mathematical Theory of Communication"gave birth to the twin disciplines of information theory and coding theory. Coding theory is the study of the properties of codes and their fitness for a specific application. Codes are used for data compression, cryptography, error-correction and more recently also for network coding. Codes are studied by various scientific disciplines, such as information theory, electrical engineering, mathematics, and computer science for the purpose of designing efficient and reliable data transmission methods. This typically involves the removal of redundancy and the correction (or detection) of errors in the transmitted data.
There are essentially two aspects to Coding theory:

2.1 Data compression (or, source coding)

The aim of source coding is to take the source data and make it smaller. Data compression or source coding is the process of encoding information using fewer bits (or other information-bearing units) than an decoded representation would use, through use of specific encoding schemes.

2.2 Error correction (or, channel coding)

Here in this chapter our main interest is in discussing error correcting codes. The aim of channel coding theory is to find codes which transmit quickly, contain many valid codewords and can correct or at least detect many errors. Coding theory with applications in computer science and telecommunication, error detection and

correction or error control are techniques that enable reliable delivery of digital data over unreliable communication channels. Many communication channel are noisy, hence there is always a chance that error can introduce during transmission. Error detection techniques allow detecting such errors, while error correction enables reconstruction of the original data. We limit our attention to binary error correcting block codes, where the "block"denotes fixed message size. Here two operations are performed on a message:

- An *encoder* converts the message m into a *codeword*, and
- the *decoder* attempts to convert the received *codeword* into original message m, where m is our fixed size binary message block.

The general idea for achieving error detection and correction is to add some redundancy (i.e., some extra data) to a message, which receivers can use to check consistency of the delivered message, and to recover data determined to be erroneous. For every scheme, detection of error is always greater than(or, equal to) correction of errors. Each encoder/decoder scheme can handle a set of predefined error conditions. Every scheme has certain limit of error correction and detection. Error-detection and correction schemes can be of two types:

- **Symmetric:** In a symmetric scheme, the transmitter sends the original data, and attaches a fixed number of check bits (or parity data), which are derived from the data bits by some deterministic algorithm. If only error detection is required, a receiver can simply apply the same algorithm to the received data bits and compare its output with the received check bits; if the values do not match, an error has occurred at some point during the transmission. Since the receiver does not have to ask for retransmission of data, it is also called as forward error correction (FEC) code.
- **Non-symmetric:** The original message is transformed into an encoded message that has at least as many bits as the original message.

We are going to use following notation throughout the chapter for discussing error correcting codes.

k	Number of information or message bits
r	Number of parity check bits
n	code length $n = r + k$
u	information bit vector, $u_0, u_1, ..., u_{k-1}$

p parity check bit vector, $p_0, p_1, ..., p_{k-1}$

s syndrome vector, $s_0, s_1, ..., s_{k-1}$

Definition 2.2.1 ***(Block Codes)*** *A block code of size M over an alphabet with q symbols is a set of M, $q-$array sequence of length n called codewords. If $q = 2$, the symbols are bits and code is called binary.*

Definition 2.2.2 ***(Hamming Weight)*** *The Hamming weight of a string is the number of symbols that are different from the zero-symbol of the alphabet used.*

Definition 2.2.3 ***(Hamming distance)*** *In information theory, the Hamming distance between two codewords of equal length is the number of positions at which the corresponding symbols are different. If codewords are of finite length than hamming distance will be of finite length too.*

Definition 2.2.4 ***(Minimum distance)*** *Given a code C, if d is the minimum of hamming distances between all codewords, then d is called minimum distance and denoted as $d(C)$.*

A code C can correct t errors if and only if $d(C) \geq 2t + 1$. In this chapter we are going to discuss *linear codes* in details because these codes are used in code based cryptography. We will also discuss *Goppa code*, in detail because we are going to use this code for implementation of our cryptosystem.

2.2.1 Linear code

A *linear code* is an error-correcting code for which any linear combination of codewords is another codeword of the code. A *linear code* of length n and rank k is a linear subspace C with dimension k of the vector space F_{q^n}, where F_q is a finite field with q elements. Any subspace of the field satisfy the property of linear codes. Such a code with parameter q is called a $q-ary$ code.

Definition 2.2.5 ***(Generator Matrix)*** *Matrix formed by using all the basis vectors of subspace C, is called generator matrix. If $v_1, v_2, ..., v_k$ is a basis for (n, k) code C, then any codeword can be written as a linear combination of these basis vectors. then generator matrix can be written as*

$$\begin{pmatrix} v_1 \\ v_2 \\ : \\ : \\ v_k \end{pmatrix}$$

A code is often represented by describing its generator matrix. To compute a generator matrix of a given code C of length n, first determine basis for the code as a vector space over field F_q. Then put these vectors in the form of a $k \times n$ matrix, where k is the dimension of vector space C over F_q.
For a code $C \in F_{q^m}$, the code $C^{\perp} = \{x \in F_{q^m} : \sum_{i=0}^{n} x_i c_i = 0; \forall c_i \in C\}$ is called *dual code* of C.

Definition 2.2.6 ***(Parity Check Matrix)*** *A parity check (H) matrix of a linear code (C) is a generator matrix of its dual code ($C^{\perp}$). Hence we have the relation $GH^T = 0$, where , G is the generator matrix of code C.*

Definition 2.2.7 ***(Syndrome)*** *Given the received vector $y = c+e$ (c is a codeword and e is error pattern) and a parity check matrix H, the syndrome of y is $S = Hy^T$.*

Observe that,

$$S \;=\; Hy^T \;=\; H(c+e)^T \;=\; Hc^T + He^T \;=\; He^T$$

c is a codeword hence its syndrome will be equal to zero. i.e., the syndrome depends only on the error pattern e and not the transmitted codeword c.

2.2.2 Decoding a linear code

Here we are going to discuss following two important decoding method for linear codes:

Definition 2.2.8 ***(List Decoding)*** *An algorithm which, for a given code C and a received word y(a vector), outputs the list $L(y|C,T)$ of all codewords at distance T apart*

$$L(y|C,T) \;=\; \{c \in C : d(y,c) \leq T\}$$

is called a list decoding algorithm with decoding radius T

For a code C with minimum distance d, a list decoding with decoding radius $T = \lfloor \frac{d-1}{2} \rfloor$, is called *bounded distance decoding*. Note that in this case list $L(y|C,T)$, is either empty or consist of single code vector.

Definition 2.2.9 ***(Maximum likelihood decoding)*** *For a given code C if received message vector is $x \in C$, then it gives the word which have least hamming distance from y, means which is most likelihood word.*

Definition 2.2.10 ***(Bounded Decoding)*** *Bounded decoder is a probabilistic Turing machine, which takes as an input a matrix/syndrome pair and outputs a string of binary elements*

2.3 Binary Goppa Code

Parameters needed for binary Goppa codes are m, n and t, where m is degree of extension, n is length of the codeword and t is size of error which can be detected and corrected by binary Goppa code. Our n should be less than equals to 2^m and $t < n/m$. In our implementation program we have used $n = 2^m$. A binary Goppa code is denoted as $\Gamma(L, g)$. A binary Goppa code $\Gamma(L, g)$, is defined by using a support L and an irreducible monic polynomial $g(z)$ of degree t, in $\mathbb{F}_{2^m}[z]$, called generator. L is defined as an ordered subset $\{\alpha_1, \alpha_2, ..., \alpha_n\}$ of $\mathbb{F}_{2^m}$ with cardinality n. This code consists of all words $\{a_1, a_2, ..., a_n\}$ in the field $\mathbb{F}_2^n$, such that

$$R_a(z) = \sum_{j=1}^{n} \frac{a_j}{z - a_j} mod(g(Z)) = 0 \tag{2.1}$$

This code is linear has dimension(k) greater than equals to $n - tm$ and minimum hamming distance will be $2t + 1$, hence it can correct t errors. The set of all binary Goppa code, with support L of cardinality n and generator polynomial g over $\mathbb{F}_{2^m}$ is denoted by $\zeta_{m,n,t}$.

Decoding of Binary Goppa Code

In this section we will give theoretical background of decoding a binary Goppa code, which will be used in the implementation part. Our implementation section will use this theory to write the program and its pseudo code for coding and decoding. We will use Patterson algorithm for decoding the code.
Let we are going to send the message x through the channel then if received word is y, then y can be written as sum of $x + e$, where e is the error in the message which can be corrected if its hamming weight is less than equals to t. Since the received word $y = x + e$ mod $(g(z))$. The syndrome of the received word, with reference to equation 2.1 can be written as,

$$s(z) = \sum_{j=1}^{n} \frac{y}{x - \alpha_j} \ mod(g(z)) = \sum_{j=1}^{n} \frac{x + e}{x - \alpha_j} \ mod(g(z))$$

$$= \sum_{j=1}^{n} \frac{x}{x - \alpha_j} \ mod(g(z)) + \sum_{j=1}^{n} \frac{e}{x - \alpha_j} \ mod(g(z)) = \sum_{j=1}^{n} \frac{e}{x - \alpha_j} \ mod(g(z))$$

Definition 2.3.1 *(**Error Locater Polynomial**) A polynomial which locates the error positions of binary Goppa code is called error locator polynomial. If* $j_1, j_2, ...j_t$

are the t locations of errors(e) of our binary Goppa code $\Gamma(L, g)$*, then error locator polynomial* $\sigma_e(z)$ *is defined as*

$$\sigma_e(z) = \prod_{i=1}^{t}(z - \alpha_{j_i})^{e_{j_i}}$$

were e is the error. We define $\sigma'_e(z)$ *as the derivative of* $\sigma_e(z)$*.*

To recover the error locator polynomial we have to solve the key equation:

$$s(z)\sigma_e(z) = \sigma'_e(z) \tag{2.2}$$

Remark: Equality can be easily checked using some general example. We can calculate the error polynomial by splitting it into even polynomial and odd polynomial parts

$$\sigma_e(z) = u(z)^2 + z.v(z)^2$$

Derivative of $\sigma_e(z)$ will be equal to

$$\sigma'_e(z) = v(z)^2$$

After substituting $\sigma_e(z)$ into the equation (2.2), we finally determine the following equation which need to be solved to determine the error locator polynomial:

$$s(z)(u(z)^2 + z.v(z)^2) = \sigma'_e(z)\ mod(g(z)) = v(z)^2\ mod(g(z)) \tag{2.3}$$

If we take $T(z) = s(z)^{-1}$, then

$$(z + T(z))v(z)^2 = u(z)^2\ mod(g(z)) \tag{2.4}$$

We can also replace the $\sqrt{(z + T(z))}$ by $S(z)$, then

$$S(z)v(z) = u(z)\ mod(g(z)) \tag{2.5}$$

Decoding of binary code uses following steps:

1. Computation of syndrome $s(z)$ for the word y.
2. Computation of inverse of the syndrome, denoted as $T(z)$.
3. computation of $S(z) = \sqrt{T(z) + z}$
4. Computation of $u(z)$ and $v(z)$, by using the equation $S(z)v(z) = u(z)\ mod(g(z))$.
5. Computation of locator polynomial $\sigma_e(z) = u(z)^2 + z.v(z)^2\ mod(g(z)$.
6. Finding the root of the equation $\sigma_e(z)$
7. Returns the error position.

2.3.1 An approach to solve key equation

We are going to Patterson's Algorithm algorithm to solve the key equation, to calculate the error locator polynomial. N. J. patterson gave this algorithm in [16]. Given $R(z)$ and $g(z)$ in $\mathbb{F}_{2^m}[z]$, where $g(z)$ is generator polynomial of degree t, we have to find the error locator polynomial ($\sigma(z)$) of degree t such that:

$$R(z)\sigma(z) = \sigma'(z) = \frac{d}{dz}(\sigma(z))$$

Take $\sigma(z) = \sigma_0(z)^2 + z\sigma_1(z)^2$, hence $\frac{d}{dz}(\sigma(z))$ will be equals to $\sigma_1(z)^2$. Now we have

$$(1 + zR(z))\sigma_1(z)^2 = R(z)\sigma_0(z)^2 \mod g(z)$$

Now we can see that $g(z)$ is irreducible polynomial of degree t, and $R(z) \in \mathbb{F}_{2^m}[z]$, hence we can have a polynomial $h(z) = z + R^{-1}(z)$. Now we have

$$h(z)\sigma_1(z)^2 = \sigma_0(z)^2 \mod g(z)$$

The mapping $f(z) \mapsto f(z)^2 mod(g(z))$ is linear and bijective over the field $\mathbb{F}_2^{tm}$, so we will have a unique polynomial $S(z)^2 = h(z) mod(g(z))$.

Now we have

$$S(z)\sigma_0(z) = \sigma_1(z)$$

The polynomial $\sigma_0(z)$ and $\sigma_1(z)$ are unique solution of the equations, which are called key equations:

$$\begin{aligned} S(z)\sigma_0(z) &= \sigma_1(z) \\ deg\sigma_0 &\leq t/2 \\ deg\sigma_1 &\leq (t-1)/2 \end{aligned}$$

Both $\sigma_0(z)$ and $\sigma_1(z)$ are computed using Extended Euclidean algorithm. Detailed algorithm and pseudo code we will discuss in implementation.

2.3.2 Root Finding of Error Locater Polynomial

Computing the root of the error locator polynomial in decoding of binary Goppa code is most time consuming and complex step. Hence it should be our initial motivation to investigate the possible paths to make the step faster. This problem is similar of many equivalent classes codes. Currently many approaches are available for root finding problem in binary extension field, their efficiency and complexity depends on the degree of extension m and degree of generator polynomial t. We are going to discuss following approaches:

1. **Chien search:** It is a fast algorithm for determining roots of polynomials defined over a finite field. When implemented in hardware, this approach significantly reduces the complexity, as all multiplications consist of one variable and one constant, rather than two variables as in the brute-force approach. This approach is used when our degree of extension m is small. The most typical use of the Chien search is in finding the roots of error-locator polynomials encountered in decoding Reed-Solomon codes and BCH codes.
   Let we have the polynomial (whose roots we have to determine) $p(x) = a_0 + a_1x + ... + a_tx^t$, with coefficients in $\mathbb{F}_{2^m}$. Let α be a generator of the multiplicative group $\mathbb{F}^*_{2^m}$
   Now $p(\alpha^i)$ will be :

$$p(\alpha^i) = a_0 + a_1(\alpha^i) + ... + a_t(\alpha^i)^t$$

   similarly

$$p(\alpha^{i+1}) = a_0 + a_1(\alpha^i)\alpha + ... + a_t(\alpha^i)^t\alpha^t$$

   Now set $a_{i,j} = a_j(\alpha^i)^j$. It is easy to obtain since we have that $a_{i+1,j} = \alpha^j a_{i,j}$. Hence if $\sum_{j=0}^{t} a_{i,j} = 0$, the α_j is root of $p(x)$

2. **Berlekamp Trace Algorithm:** This algorithm was originally published in [1]. This is a very efficient algorithm for finite field with small characteristic.

   Definition 2.3.2 *(**Trace:**) The trace function $Tr()$ of elements of $\mathbb{F}_{2^m}$ over $\mathbb{F}_2$ is defined as*

$$Tr(z) = z + z^2 + z^{2^2} + ... + z^{2^{m-1}}$$

   It maps the field $\mathbb{F}_{2^m}$ over its ground field $\mathbb{F}_2$. A key property is that if $\{\alpha_1, ...\alpha_m\}$ is basis of the extension field $\mathbb{F}_{2^m}$, then every element $\alpha \in \mathbb{F}_{2^m}$ can be uniquely represented as binary m-tuple of the following form:

$$(Tr(\beta_1\alpha), ..., Tr(\beta_m\alpha))$$

   $\{\beta_1, \beta_2, ..., \beta_m\}$ is any basis of $\mathbb{F}_{2^m}$ over $\mathbb{F}_2$. The main idea of BTA algorithm is that any $f(z) \in \mathbb{F}_{2^m}[z]$, with $f(z)|(z^{2^m} - z)$, splits into two polynomials

$$g(z) = gcd(f(z), Tr(\beta.z)) \text{ and } h(z) = gcd(f(z), 1 + Tr(\beta.z))$$

   Will see its pseudo code in chapter implementation of next part.

3. (**Zinoviev method:**)This method was first published in [20]. This procedure used to find the roots of the polynomial of degree lesser or equal to 10. Zinoviev methods find the monic affine multiple of smallest degree of f of degree $d \geq 0$ and lesser that 10 [20]. At step $i \geq 0$, we compute multiple of f of degree $2^{\lceil log_2 d \rceil + 1}$ and we try to decimate the non linear terms by solving a system of homogeneous equations. If system has no solution we go $i + 1$ step. Besides, an algorithm proposed by Berlekamp, Rumsy and Solomon in [2] ensue to find a affine multiple degree 2^{d-1} and thus guarantee Zinoviev method terminates, in the worst case at step $d - 1 - \lceil log_2 d \rceil$. After that finding the roots of affine polynomial is quite easier than general case. There we just have to solve linear system of order m over $\mathbb{F}_2$, then we have to determine the roots of f among the roots of affine polynomial we have found.

Definition 2.3.3 *(**Linearized Polynomial**) A polynomial over $\mathbb{F}_{q^m}$ of the form:*

$$L(x) = \sum_{i=0}^{n} l_i . x^{q^i}$$

is called linearized polynomial, where $l_i \in \mathbb{F}_{q^m}$ and $l_n = 1$

***Definition 2.3.4** (**Affine Polynomial:**) If L is linearized polynomial over $\mathbb{F}_{q^m}$ and $c \in \mathbb{F}_{q^m}$, then affine polynomial $A(x)$ will have the form:*

$$A[x] = L[x] + c$$

.

But in our case we are using $q = 2$. The *trace polynomial* is example of linearized polynomial.
Let we have affine polynomial $A(x) = L(x) + c$

$$A(x) = \sum_{i=0}^{n} l_i . x^{q^i} + c$$

Chapter 3

Code Based Cryptography

We have already given a brief description of cryptography in introduction. Here we are going to discuss code based cryptosystems. We are going to do implementation of following two cryptosystems in out project work.

3.1 McEliece cryptosystem

The McEliece cryptosystem is an *asymmetric* encryption algorithm. This cryptosystem was developed in 1978 by Robert J McEliece [14]. This was the first scheme in public key cryptography to use randomization in encryption. This cryptography system is not being used frequently by the cryptographers but is a member of "Post Quantum Cryptography", as it is immune to attack by quantum computes using shor's algorithm [18]. This system has *good security reduction*, is efficient but have large public key size. That's why encryption and decryption both uses lengthier procedure. There are several attempts made to reduce the key size to standard level without compromise in security, following P. Gaborits papers in 2005[8]. The algorithm is based on the hardness of decoding a general linear code (which is known to be NP-hard). To generate the key for this cryptosystem, we have to select a particular error-correcting-code for which an efficient decoding algorithm is known, and which is able to detect and correct the t-errors. The original algorithm uses binary Goppa codes, though the original parameters are proved to be weak [3].

3.1.1 Scheme Description

Similar to any public key cryptography scheme, McEliece scheme also consists of three algorithms: a probabilistic key generation algorithm which produces a public

and a private key, a probabilistic encryption algorithm, and a deterministic decryption algorithm. Common security parameters are n,k and t which are shared by all the users.

Key generation

The objective of this algorithm is to generate the set of keys that will be used for encryption(uses public key) and decryption(uses private key).

1. Alice selects a (n,k)-binary Goppa code Γ capable of correcting t errors. This code must possess an efficient decoding algorithm.
2. A $k \times n$ generator matrix G is generated by Alice for the code Γ.
3. Alice selects a random $k \times k$ binary non-singular matrix S.
4. Alice selects a random $n \times n$ permutation matrix P.
5. Alice computes the $k \times n$ matrix $\hat{G} = SGP$.
6. Alices public key is $(\hat{G}, t)$; her private key is (S, G, P).

Encryption

Suppose Bob wishes to send a message m to Alice whose public key is $(\hat{G}, t)$:

- Bob encodes the message m as a binary string of length k.
- Bob computes the vector $c' = m\hat{G}$.
- Bob generates a random n-bit vector e containing exactly t ones (a vector of hamming weight t, with length n)
- Bob computes the ciphertext as $c = c' + e$.

Decryption

After receiving the ciphertext c Alice uses the following steps to decrypt the cypher text:

- Computes the inverse of P, P^{-1}.
- Alice computes $\hat{c} = cP^{-1}$.

- Uses the decoding algorithm for the code C to decode $\hat{c}$ to $\hat{m}$.
- Alice computes $m = \hat{m}S^{-1}$.

3.1.2 Theory behind the Scheme

For each irreducible polynomial $g(x)$ over F_{2^m}(where m is degree of field extension) of degree t, there exits a binary irreducible Goppa code of length $n = 2^m$ and dimension $k \geq n - mt$, which is capable of correcting at most $t - errors$. Because of being a linear code, it can be described by its generator matrix G, of size $k \times n$. As given in key generation algorithm, we also need a non-singular $k \times k$ binary matrix and a permutation matrix P of size $n \times n$. We can now calculate our $\hat{G}$, which hides the structure of generator matrix G and used as a public key.

$$\hat{G} = SGP$$

Now we have two pair of keys: public key $(\hat{G})$ and secret key (S, G, P). It is hard to calculate the generator matrix G, using $\hat{G}$, without knowing its structure. Encryption algorithm consists of multiplication of the message m(in the form n bit binary array) by $\hat{G}$ and addition of a error vector e of hamming weight t with modulo 2(because we are working with binary field). Hence after encryption stage Alice receive the ciphertext c

$$c = (m.\hat{G} \oplus e)$$

To decrypt the ciphertext c, at first Alice multiply c with P^{-1}, Hence we receive

$$\hat{c} = c.P^{-1} = (m.\hat{G}.P^{-1} + e.P^{-1}) = m.S.G \oplus e'.$$

We have replaced eP^{-1} by e'

After using the decoding algorithm we receive $\hat{m} = m.S$. By multiplying S we can receive the message m in the form of binary arrays, which is our original plain text.

3.1.3 Security and attacks

Due to hardness of decoding of general linear code this cryptosystem is considered to be fairly secure. Variants of this cryptosystem exist, using different types of codes. Most of them were proven less secure; they were broken by structural decoding. McEliece with Goppa codes has resisted cryptanalysis so far. Recently used two main approaches to attack by cryptanalysis are given below:

Structural attacks These attacks are also knows as *key attacks*. In these attacks cryptanalyst try to reconstruct a decoder for the code generated by the public-key $\hat{G}$ by studying its structure. From the very construction of the system, the code generated by the public-key $\hat{G}$ is equivalent to Γ. The best known attacks consist in enumerating the codes in the family to find a code which is equivalent to Γ[12, 13, 7].

Decoding attacks Also knows as *message attacks*. Since Γ is equivalent to the code generated by the matrix $\hat{G}$, both codes have the same error-correcting capability. The cost of the attack depends on the parameter of the code, its length, its dimension and its error correcting capability. It implies that the parameters of the system have to be chosen carefully. A recent paper describes both an attack and a fix [3]. Another paper shows that for quantum computing key sizes must be increased by a factor of four due to improvements in information set decoding [3].

3.2 Niederreiter cryptosystem

The Niederreiter cryptosystem is a variation of the McEliece Cryptosystem developed in 1986 by Harald Niederreiter [15]. Niederreiter is equivalent to McEliece from a security point of view. Similar to McEliece this cryptosystem also uses binary Goppa code for key generation. This cryptosystem uses parity check matrix for key generation in place of generator matrix. In this cryptosystem syndrome is used as ciphertext and message as an error pattern. Let H be an $(n-k)\times n$ parity check matrix for the code C, M any $(n-k)\times(n-k)$ randomly generated non-singular binary matrix and P is a random permutation. The encryption of Niederreiter is about ten times faster than the encryption of McEliece. The system is described below:

Key generation

- Alice calculate $H' = MHP$.
- Alice use (H', t) as public key and (H, M, P) as secret key.

Encryption

Our message will be n bit binary array of hamming weight t. Our ciphertext $c = H'm^T$.

Decryption

Alice receive the ciphertext c and does the following:

- Alice computes $M^{-1}c = HPm^T$.
- Alice applies a syndrome decoding algorithm to recover Pm^T.
- Alice computes the message m via $m^T = P^{-1}Pm^T$.

Recommended values for these parameters are n = 1024, t = 38, k = 644. Niederreiter's original proposal was broken[19], but the system is secure when used with a binary Goppa code.

3.3 The Hybrid McEliece Encryption Scheme (HYMES)

Here, we will describe a slightly modified version of the scheme (which we call hybrid). As we have already stated that this scheme has two important modification. Two modifications are, the first increases the information rate by putting some data in the error pattern. The second reduces the public key size by making use of a generator matrix in row echelon form. Here we will describe the system after these modifications. Similar to original McEliece cryptosystem, here also we choose a generator polynomial $g(x) \in F_{2^m}[x]$ of degree t and support $L = \{\alpha_1, \alpha_2, ..., \alpha_n\}$. We construct the generator matrix G for the Goppa code $\Gamma(L, g)$. Later as we have told before, we change generator matrix G in its reduced row-echelon form. Hence we can write $G = [Id|R]$. Here Id will be the identity matrix. Here R will be our public key and $(L,\ g)$ will be secret key.
Let m and t are two integers then $n = 2^m$ and $k = n - mt$.

Key generation

- Bob selects a random binary $(n,\ k)$-Goppa code, $\Gamma(L, g)$ which can correct t errors and for which an efficient decoding algorithm exist.
- Bob generates the a matrix R of size $k \times (n - k)$ such that $(Id|R)$ is a row echelon form of generator matrix.
- We permute the columns of matrix R using a permutation array $Perm$, which is generated by column permutation used during the transformation of generator matrix into reduced row echelon form.
- Public Key=$\hat{R} = Perm(R)$ and Secret Key =$(L,\ g)$

Encryption

- Encodes the message(x) into binary string of size k.
- Encodes the error(e) into binary string of size n and hamming weight t.
- Sends the ciphertext $y = (x|x\hat{R}) + e$ to Bob.

Decryption

- Use efficient decoding algorithm to correct the t errors and to calculate $y' = (x|xR)$.
- Calculates the y using y'.

Cryptographic security

The first reductional proof of security for the McEliece encryption scheme was given by Kobara and Imai in [10]. In the same paper, several semantically secure conversions, generic and ad-hoc, are proposed. The purpose of those conversion is to transform a One Way Encryption (OWE) scheme, the weakest notion of security, into a scheme resistant to adaptive chosen ciphertext attack (IND-CCA2), the strongest notion of security (in the random oracle model).

Part II

Implementation

Chapter 4

Implementation of HYMES in "C"

We have the complete project in C language. This program can easily run on linux based systems without any changes. People are allowed to make any changes in the program for the research purpose. We can characterize this project in four sections, *Library functions*, *Key-generation*, *Encryption and Decryption*. Here we are describing each one in detail.

4.1 Library Function

In this section we define Galois field, polynomial over Galois field and matrix properties. We divide this section in following subsections.

Parameter

We will make a library file param.h, which contains the parameters, needed for our source code. We need two parameters for the source code, first degree of extension(m) and second error weight(t). Here we have defined degree of extension as LOG_LENGTH and error weight as ERROR_WEIGHT. We have restricted our value of LOG_LENGTH to be less than 16. Hence we have initially taken LOG_LENGTH=11 and ERROR_WEIGHT=30, which can be changed later.

Size

To define the size of different variables used, we make a library field *size.h*. This file includes the *param.h* in which we define the following variables and functions.

- Degree of extension m
- Maximum number of errors, which could be corrected by our Goppa Code t
- Length of the code word in the Goppa code will be less than equal to 2^m. But here we are choosing of code of length 2^n.
- CODIMENSION=EXT_DEGREE*NB_ERRORS=m*t. This function gives co-dimension of the generator matrix as defined in Goppa code.
- DIMENSION = LENGTH-CODIMENSION=n-m*t. This function gives dimension of the Goppa code.
- SECRETKEY_BYTES =(LENGTH * sizeof (long) * BITS_TO_LONG(CODIMENSION) + (LENGTH + 1 + (NB_ERRORS + 1) * NB_ERRORS) * sizeof (gf_t). This function gives the number of bytes needed for storing secret key.
- PUBLICKEY_BYTES =(BITS_TO_LONG(CODIMENSION) * sizeof(long) * DIMENSION). This function gives number of bytes needed for storing public key. Our public key is the generator matrix $\hat{G}$, which have the dimension equals to CODIMENSION*DIMENSION.
- CIPHERTEXT_BYTES BITS_TO_BYTES(LENGTH). This function gives number of bytes needed for the ciphertext and same number of bytes would be needed for message.

Galois Field

In this section we are going to discuss the implementation part for the properties of field F_{2^m}, which is an extension field of degree m, over a binary field. If $P(X)$, is the irreducible polynomial of degree m, then

$$F_{2^m} \approx \frac{F_2[X]}{P(X)} \tag{4.1}$$

Field F_{2^m}, is collection of all possible combination of 0s and 1s of length m. If α is a primitve element of this field then every element(except 0), can be generated by α. Hence $F_{2^m} = \{0, \alpha, \alpha^2, ..., \alpha^{2^m-1} = 1\}$. We know that 2 is always primitve element for binary extension field, hence to make things easily computable, here we are going to use $\alpha = 2$. We are going to implement our code in 'C'. We know that our computer stores everything in binary form. Hence every element of field F_{2^m}, can be represented in form of some integer. We made two library files gf.c and gf.h, which uses the every property of the Galois field elements. gf.c file includes the

library file gf.h.Here we will start with definition of Galois field elements (see section 1.2). Here we are interested in binary field. The usual XOR operation serves the purpose of addition. However for the extension field $\mathbb{F}_{2^m}$, we needed the operations defined Galois field element is denoted as gf_t here. Firstly we define type of the gf_t, its type would be unsigned short which have size 2 bytes. gf.c file includes the library file gf.h.
Here we take maximum degree of extension= 16. We are taking maximum degree of extension to be 16, because we have defined our Galois field element of type unsigned short, which is of size 2 bytes (16 bits), hence it can store only 16 bit binary number in extension field.
Our implementation needs irreducible polynomial of m(degree of extension) hence we store these polynomial in an array. Our array would be of size *maximum extension degree*. This array contains the decimal value of the irreducible polynomial of degree m, where $2 \leq m \leq 16$.
In Galois field library we include the following properties of the Galois field:

- **Exponentiation:**If α is generator of the Galois field $\mathbb{F}_{2^m}$ then any element x of the field can be written in the form of α^i, where i is unique for every x. It is called exponential representation of the field element x. Here we already choose exponential value of 0 equals to 0. Pseudo code for the calculation of exponential value is given in the section 5.1

- **Logarithm:** This properties inverse of the exponentiation. If α^i is a Galois field element of the field $\mathbb{F}_{2^m}$ then it will have a logarithmic value x less than equal to m.

- **Multiplication, Division and Inverse:** If $x = \alpha^i$ and $y = \alpha^j$ are two Galois field elements then their multiplication will be addition of their power. Hence

$$Mult(x, y) = x.y = \alpha^i \alpha^j = \alpha^{i+j}$$

From the property of primitive element, $\alpha^{2^m-1} = 1$, hence if $|i + j| \geq 2^m - 1$, then

$$Mult(x, y) = \alpha^{((i+j) \ mod(2^m-1))}$$

Similarly we can define division in following way:

$$Div(x, y) = x.y = \alpha^i/\alpha^j = \alpha^{i-j} \tag{4.2}$$

if $|i + j| \geq 2^m - 1$, then

$$Div(x, y) = \alpha^{((i-j) \ mod(2^m-1))}$$

If we take $x = 1$ then $Div(x, y) = Inv(x)$.

Note:

Pseudo codes regarding the properties of Binary Galois field are given in appendix **A.1**.

Polynomial

During the implementation of the cryptosystem we need to represent the field in the form of polynomial. We also need to calculate irreducible polynomial over our Galois field $\mathbb{F}_{2^m}$. Here we take the coefficient of elements as field elements of Galois field. We need to use the properties of Galois field hence here we include library functions of the Galois field also. We define our structure of polynomial using coefficients, size and degree. Here size of the polynomial will always be greater than or equal to the degree of polynomial. We use the following properties of the polynomials in our program:

- **Degree:**Degree of the polynomial $P(x)$ is defined as the greatest power of x, which have non-zero coefficient in the polynomial.

- **Addition:** Here addition of the polynomial is addition of coefficient of same degree in Galois field. Resulting polynomial will have the size equal to maximum of the size of the polynomials. We can calculate the degree of this polynomial using the above definition.

- **Multiplication:** If there are two polynomial $p(x) = \sum_{j=0}^{k_1} p_j x^j$ and $q(x) = \sum_{i=0}^{k_2} q_i x^i$ then multiplication of the polynomials will be:

$$r(x) = p(x) * q(x) = \sum_{l=0}^{k_1} r_l x^l$$

  where $r_l = \sum_{l=i+j} p_i q_j$.

- **Division:** Let there are two polynomials $p(x)$ and $q(x)$. If we have to calculate $p(x)/q(x)$, then division have to be done using algebraic methods. Let degree of polynomial p is less than degree q then its remainder will be equal to $p(x)$ and quotient will be equal to 0. Remainder of the polynomial will be of degree $degree(p) - 1$ and quotient will be of degree equals to $degree(p) - degree(q)$.

- **Inverse:** Inverse of the polynomial in the field $\mathbb{F}_{2^m}[x]$ is done using irreducible polynomial of degree t.

- **Squaring of polynomial:** Here we will also do the squaring of polynomial with respect to the irreducible polynomial $g(x)$ of degree t. Square function is implemented with all previously computed value x^{2i} $modulo(g(x))$, later we adjust coefficients. Here we simply square the first $t/2-1$ coefficient and later coefficients are squared using the precomputed values of z^{2i} $modulo$ $g(z)$.

- **Square root:** Calculation of square root is done with the help of the following equation:

$$z^{2^{mt-1}} = \sqrt{z}\ mod(g(z))$$

  Hence we have to compute all $z^{i/2}$ $mod(g(z))$ for all odd value of i less than and equal to $g(z)$.

- **Degree of smallest factor of polynomial:** Calculation of degree of smallest factor of polynomial help us to check the irreducibility of the polynomial.

- **Irreducibility Check** We have seen that in the implementation of the program we need to generate an irreducible polynomial $g(z)$ of degree t in $\mathbb{F}_{2^m}[z]$. Irreducibility check depends on the definition that for a irreducible polynomial $g(z)$, there exist no $i < t$ such that $g(z)|(z^{2^{mi}} - z)$ or $z^{2^{mi}} = z\ mod(g(z))$.
  This implies that if $g(z)|(z^{2^{mi}} - z)$ then $g(z)$ have a factor of degree i hence as given above if degree of smallest factor of $g(z)$ is less than t then polynomial is reducible.
  Here we solve $z^{2^{mi}}$ $mod(g(z))$ for all $i \leq t/2$. If these equation do not give any result equals to z then polynomial $g(z)$ will be an irreducible polynomial.

Note

Pseudo codes regarding the properties of Binary Galois field are given in appendix **A.2**

Matrix

As we have discussed in the theory section that matrix property help us highly in key generation part. Hence here we are going to discuss the matrix structure and its properties. To design a matrix structure we need to have two inputs variable first number of columns and second number of rows. To make the computation faster we will design a structure in which this matrix will store the bits in the form of blocks of 8*sizeof(unsigned long), which is called as a word and the length is define as BITS_PER_LONG.
Here we are discussing following properties of the matrix.

- **Addition:**Addition of two matrix is possible if and only if dimension of both matrices are equal. Addition will be binary addition of its elements. If there are two matrix $A = [a_{ij}]$ and $B = [b_{ij}]$ then addition is defined as

$$C = A + B = [(a_{ij} + b_{ij})\ mod\ 2]$$

- **Matrix Multiplication:** Multiplication of two matrix $A_{l\times m}$ and $Bm \times n$ is possible if only if number of rows of B is equals to number of columns of B. If $C_{l\times n} = A.B$ then its coefficient will be

$$Ci, j = \sum_{k=0}^{k=m} a_{i,k}.b_{k,j}$$

- **Row echelon form** Definition of the row echelon form is given in the section (1.5). By using elementary row operations we change any matrix into its row echelon form. Here in our program when we change a matrix in its row echelon form using *Gaussian Elimination* method then we also receive a permutation array which would be used in permuting the generator matrix.

Note:

Pseudo codes regarding the properties of Binary Galois field are given in appendix **A.3**

4.2 Keypair

Reference binary Goppa code. We have two input parameters m and t, where m is the degree of extension and t is the number of errors for the Goppa code. Hence we can calculate the length $n = 2^m$ and co-dimension $r = mt$ and dimension $k = n-mt$. We denote binary irreducible Goppa code as $\Gamma(L, g)$, where L is called support and g is defined as generator polynomial.

- Support $L = (\alpha_1, \alpha_2, ..., \alpha_n)$ distinct in F_{2^m} and
- $g(x) \in F_{2^m}[x]$ monic irreducible polynomial of degree t.

Let $\{\beta_1, \beta_2, ..., \beta_m\}$ is a simple basis of vector space F_{2^m} over the binary field. Then any $\gamma \in F_{2^m}$ can be uniquely written as

$$\gamma = a_1\beta_1 + a_2\beta_2 + ... + a_m\beta_m$$

where $\beta_i \in F_2, \forall i$. We can define parity check matrix (H) for the binary irreducible Goppa code as follows:

$$H = \begin{pmatrix} 1 & 1 & ... & 1 \\ \alpha_1 & \alpha_2 & ... & \alpha_n \\ : & : & : & : \\ \alpha_1^{t-1} & \alpha_2^{t-1} & ... & \alpha_n^{t-1} \end{pmatrix} \begin{pmatrix} g(\alpha_1)^{-1} & & \\ & \ddots & \\ & & g(\alpha_1)^{-1} \end{pmatrix}$$

Hence

$$H = \begin{pmatrix} g(\alpha_1)^{-1} & g(\alpha_2)^{-1} & ... & g(\alpha_n)^{-1} \\ \frac{\alpha_1}{g(\alpha_1)} & \frac{\alpha_2}{g(\alpha_2)} & ... & \frac{\alpha_n}{g(\alpha_n)} \\ : & : & : & : \\ \frac{\alpha_1^{t-1}}{g(\alpha_1)} & \frac{\alpha_2^{t-1}}{g(\alpha_2)} & ... & \frac{\alpha_n^{t-1}}{g(\alpha_n)} \end{pmatrix}$$

Hence binary irreducible Goppa code $\Gamma(L, g) = \{a \in F\{0,1\}^n | aH^T = 0\}$ H is a parity check matrix of (n, k)-binary Goppa code. There are t parity check equations with coefficient in F_{2^m}, thus at most $r = mt$ parity check equations with binary coefficient, because the codimension is at most mt.
We are going to use $\Gamma(L, g)$-binary Goppa code, to detect and correct at most t errors hence its minimum distance, $\text{dmin}(\Gamma(L, g)) \geq 2t + 1$. $\Gamma(L, g))$ is an alternate code of designed distance $t + 1$. Key generation scheme uses the following steps:

- Support $L \in F_{2^m}$.
- generator $g(x) \in F_{2^m}[x]$ a monic irreducible polynomial of degree t.
- Feed the parity check matrix H
- Apply Gaussian Elimination algorithm. We will get $H = S(R|I)P$, where S is a non-singular binary matrix, I is identity matrix and P is permutation matrix. We denote $\widehat{H} = (R|I)$, hence $\widehat{G} = (I|R^T)$.
- Permute the support(L) using permutation matrix(P).
- Secret Key =(g, t), Public Key R^T

Here we will show how to generate set of keys. Here we made a file keypair.c. This files contains the functions which generate public key and secret key(see the section 3.3). This file includes the library files, size.h, gf.h, poly.c and matrix.h. This file contains the following functions:

- gop_supr(int n,gf_t *L), this function calculates the Goppa support L. Initially input array L have the value $L[i] = \alpha^i$, where i varies 0 to $n-2$ and $L[n-1] = 0$. This function randomly permutes the array to give Goppa support L. u32rand is used for generating random number.

- key_genmat(gf_t *L, poly_t g), this function gives the generator matrix H for the Goppa code $\Gamma(L,\ g)$. This matrix will be of the dimension $r \times n$, where $r = (numberoferrors).(extensiondegree)$ and $n = 2^{ext_deg()}$. Firstly we put $H = 0$, means every element of the matrix of H equals to zero. Now our first will be to calculate $x = g(L[i])$, later we will calculate inverse of x that we will store in x itself hence our command line for that will be $x = gf\_inv(x)$. We also take one more variable y which will store the value of $x = \frac{1}{g(L[i])}$. Our x is in the field $\mathbb{F}_{2^m}$, which is of size EXT_DEGREE bits, hence we will expand the x in binary form and will write these element in matrix H by using the following for loops

  for(k=0;$k < EXT\_DEGREE$;k++)
  {
  if(y & (1<< k))
  mat_set_coeff_to_one(H,j*EXT_DEGREE + k,i)
  }

  Hence IF statement will check if y and 2^k have common factors then it will put 1 there. In this way we get the binary representation of y. In next step we will multiply y by $L[i]$, hence our y will be $y = L[i].y$, this step is used t number of times, where t is the NB_ERROR. The similar steps are followed for all i from 0 to $n-1$. Now we have parity check matrix H.
  Now our next target is to get the reducible row echelon for of the matrix H of size $(n-r) \times r$. Because we can write $H = SRP$, where S is singular binary matrix R is reduced row echelon generator matrix and P is permutation matrix which will be have bijective relation with the array $perm$. We use function $mat\_rref(H)$ to get the reducible row-echelon matrix R and permutation array $perm$.Here our matrix will have one $r \times r$ identity matrix and one $(r \times (n-r)$ matrix which will be in the row-echelon form. If $perm$ is nill then this function will return $NILL$, else takes a matrix $R = 0$ of size $r \times (n-r)$. This matrix takes the element of from the second matrix of H after tranformation and then permuting the row by array $perm$.(See the code). This function will return the matrix R. Step used above are as followed
  $H_{r\times n} = [I_{r\times r}|R_{r\times(n-r)}]$, $\hat{R} = perm((R)^T)$.

After generating the generator matrix R, we will generate the key-pair. pair contains public key and secret key. We save the key in binary files named as pk(public key)

and sk(secret key or personal key).

Public Key

Public key file is named as pk.bin. This file contains length of the code n, number of errors t and matrix R, returned from the function *key_genmat*.

Secret Key

Secret key file is named as sk.bin. This file contains inverse of the Goppa support L named as $Linv$, polynomial g, square root of polynomial g in the field $F_2[z]/g(z)$ and precomputed syndrome $R_c(z)$, where $c[i] = 1$ for every 'i'.
You can refer the pseudo code for the key generation in appendix **A.4**.

4.3 Encryption

Encryption of the message using HYMES scheme is very easy. Our public key for the encryption will be R. If input message is M then we divide the message into blocks of length k. Here we will also generate the error $e[\,]$ of length n and hamming weight t. Our ciphertext c will be:

$$c = [m|mR] + e[\,] \tag{4.3}$$

Pseudo code for the encryption is given in appendix **A.5**.

4.4 Decryption

Decryption of the ciphertext is done using the private key or secret key (sk). Decrypted message is divided in block size of n. We use error correcting algorithm to correct the error e in the ciphertext. The decryption process uses the Patterson Algorithm[section 2.3.1] for decoding.
There are three steps of decoding:

Syndrome Computation

To make the syndrome computation faster it would be preferable to calculate

$$f_{\alpha_j}(z) = \frac{1}{z - \alpha_j} \ \ mod(g(z))$$

for all $\alpha_j \in L$, $1 \leq j \leq n$.
This work is already done during key generation. Hence our secret key already have $f_{\alpha_j}(z)$. This will speed up the computation of syndrome. Let $c = \{c_1, c_2, ..., c_n\}$ is the received word then syndrome $R_c(z)$ will be

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

Solving the Key Equation

If $\sigma(z)$ is the error locator polynomial and $R_c(z)$ is the syndrome of word c, then key equation will be:

$$\sigma(z).R(z) = \sigma'(z) \tag{4.4}$$

we can devide $\sigma(z)$ into odd and even parts: $\sigma(z) = \sigma_0^2(z) + \sigma_1^2(z).z$. We finally determine the following equation which needs to be solved to determine error positions:

$$(\sigma_0^2(z) + \sigma_1^2(z).z).R(z) = \sigma_1^2(z) \tag{4.5}$$

$$(1 + zR(z)).\sigma_1^2 = R(z)\sigma_0^2 \ \ mod(g(z)) \tag{4.6}$$

We can put $h(z) = z + R(z)^{-1} \ \ mod(g(z))$. If $S(z)^2 = h(z) mod(g(z))$ then we can write the equation(4.5) as

$$R(z).S(z) = \sigma_0(z) \tag{4.7}$$

To solve the key equation we have to calculate the square root of z^i for all i from 0 to t.

$$T_i(z) = \sqrt{z^i} \tag{4.8}$$

We use this method to calculate the polynomial $S(z)$ from $h(z)$.

Here we use extended euclidian algorithm(section 1.6.2) to calculate the polynomials $\sigma_0(z)$ and $\sigma_1(z)$. Hence we can calculate the error locator polynomial $\sigma(z)$.

$$\sigma(z) = \sigma_0(z)^2 + \sigma_1(z)^2 z \tag{4.9}$$

Solving Error Locator Polynomial

Root finding of polynomials over finite fields is a classical algebraic algorithmic problem. It is considered as one of the most time-consuming sub-process of the decoding process of Reed-Solomon, BCH and Goppa codes. There are some well known approaches for finding roots of the so-called error-locator polynomial. The most widely known root finding algorithm is Chien search method [17], which is a simple substitution of all elements of the field into the polynomial, so it has very high time complexity for the case of large fields and polynomials of high degree. Berlekamp Trace Algorithm (BTA) [6] is another well known method. It is a recursive method based on the trace function properties.
Here we present a hybrid method involving BTA and a method proposed by Zinoviev [20]. Zinoviev proposed direct root finding procedures for polynomials with degree at most 10. Our idea is to compute directly the roots with Zinoviev procedures up to some degree and to use BTA for greater degrees. Moreover, we improve Zinoviev procedures for polynomials of degree 2 and 3 with time-memory tradeoffs. We analyze both the theoretical complexities and the experimental complexities of our proposal. We obtain a theoretical gain of 93% over Chien method and 46% over BTA. Experimental results confirm theory up to degree 4 at least. For instance with m = 11, t = 32 and dmax = 4, our method takes 60% of the total decryption time with respect to 72% for BTA and 87% for Chien. Error locator polynomial will have atmost t roots if our code can fix the t errors. The error locator polynomial will be

$$\sigma_e(z) = \prod_{i=1}^{i=n}(z - \alpha_i)^{e_i}, \quad where\ e_i \in \{0, 1\} \tag{4.10}$$

If $\sigma_e(z)$ can locate t errors then $\sum_{i=1}^{i=n} e_i = t$. We can see that if $e_i \neq 0$ then $\sigma(\alpha_i) = 0$ and vice-versa.
The efficiency of the root finding algorithms is a problem that we study in code-based cryptography. McEliece-type cryptosystems are often based on binary Goppa codes. Their decryption algorithm employs an algebraic decoding process to recover the original message from the cyphertext. The most time-consuming stage, in the implementation of algebraic decoding of binary Goppa codes, with practical parameters, is the root finding of the error locating polynomial. This polynomial fulfills the above mentioned properties.

Decryption Complexity Theoretical Complexity = number of arithmetic operations in F_{2^m} required to decrypt in the worst case.

- Syndrome computation $\mathcal{O}(nt)$
- Key equation solving $\mathcal{O}(t^2)$

- Error locator polynomial root finding
  1. Berlekamp Trace Algorithm $\mathcal{O}(mt^2)$
  2. chien's search Algorithm $\mathcal{O}(nt)$

Important step BTA algorithm is to calculate tract of an element $\alpha \in \mathbb{F}_{2^m}$.

Trace Calculation

We know that trace of z will be

$$Tr(z) = z + z^2 + ... + z^{2^m-1} \tag{4.11}$$

Tr(z) will be a polynomial in z. We can change this polynomial into a polynomial of degree t using the polynomial $g(z)$ by following way:

$$Tr(z) = \begin{vmatrix} z & mod(g(z)) \\ z^2 & mod(g(z)) \\ & \vdots \\ & \vdots \\ z^{2^m-1} & mod(g(z)) \end{vmatrix}. \tag{4.12}$$

Trace of any elements $\alpha_i \in \mathbb{F}_{2^m}$, will be

$$Tr(z) = \begin{vmatrix} \alpha^i z & mod(g(z)) \\ (\alpha^i z)^2 & mod(g(z)) \\ & \vdots \\ & \vdots \\ (\alpha^i z)^{2^m-1} & mod(g(z)) \end{vmatrix}. \tag{4.13}$$

To solve the error locator polynomial $\sigma(z)$ we use BTA algorithm as given in section(2.3.2). This algorithm gives t error poisitions. Hence now we have an error vetor e of hamming weight t and size n. If y is the cipher text then we can retrive the message in following way:

$$y' = y + e = [x|xR] + e + e = [x|xR]$$

x is our plain text here.

All the pseudo codes regarding decryption process are given in the appendix **A.6**.

Part III

Further Research Work

Chapter 5

Improvisation in Niederreiter Cryptosystem

The Niederreiter cryptosystem is a variation of the McEliece Cryptosystem developed in 1986 by Harald Niederreiter. It applies the same idea to the parity check matrix H of a linear code. Niederreiter is equivalent to McEliece from a security point of view. It uses a syndrome as cipher-text and the message is an error pattern. The encryption of Niederreiter is about ten times faster than the encryption of McEliece. In the Niederreiter scheme one important step is to encode a message m into an array of size n and hamming weight less than equal to t. In the original Niederreiter scheme, he used to concatenate $n - t$ zeros to the binary message of size t, if we are using a Goppa code which can correct t errors. Our Goppa code $\Gamma(L, g)$ can correct maximum t errors. If we use the list decoding then number of elements in the list will be $\sum_{i=0}^{t} \begin{pmatrix} n \\ i \end{pmatrix}$. But if we use one encoding method such that it gives an array of length n and hamming weight t, then size of the list will be $\begin{pmatrix} n \\ t \end{pmatrix}$. Hence if the hamming weight is fixed to t, then it would be easy to correct the errors.
Nicolas Sendrier published one encoding method for this scheme in his paper [17]. This method is known as "Constant Weight Encoding". Constant weight encoding function (ϕ) can be represent as:

$$\phi : A \rightarrow W_{(n,\ t)}$$

If A is the collection of binary array of length l and $W_{(n,\ t)}$ is collection of all the binary arrays of length n and hamming weight t, then $|A| = 2^l$ and $|W_{(n,\ t)}| =$

$\begin{pmatrix} n \\ t \end{pmatrix}$. ϕ is constant weight encoding and also it should be invertible hence,

$$|A| = |W_{(n,\ t)}|$$

$$2^l = \begin{pmatrix} n \\ t \end{pmatrix}$$

Hence required length for encoding a message should be equal to $l = log_2 \begin{pmatrix} n \\ t \end{pmatrix}$. Constant weight encoding method described by Nicolas Sendrier have linear time complexity in n. Here we are going to discuss one easier encoding method with lesser time complexity. Assume that our encoding function ϕ is such that it takes a message array of length l and gives encoded message of length n and hamming weight t.

$$\phi : A_l \rightarrow W_{(n,t)}$$

Here A_l is the collection of the message arrays of length l and $W_{(n,t)}$ is as defined before. Now our aim is to calculate the length l using our new encoding method.
Here our main aim is to find t positions for 1 in between 0 to $n-1$, defined as $\{\delta_1, \delta_2, ..., \delta_t\}$.
Our next aim will be to calculate the length l for this encoding method. This length l would be a multiple of t for our encoding method. Now we will devide the binary array of the length l in t subarrays of length x. Here we will calculate the decimal representation of all t subarrays, defined as $\{d_1, d_2, ..., d_t\}$. Now we will calculate the $\{\delta_1, \delta_2, ..., \delta_t\}$ in following way.

$$\delta_0 = 0$$

$$\delta_i = \delta_{i-1} + d_1 + 1 \qquad where 1 \leq i \leq t$$

By using the above equations we can see that

$$\delta_t = d_1 + d_2 + ... + d_t + t - 1$$

Maximum value of d_i can be equals to 2^x, hence maximum value of δ_t will be equals to

$$\delta_t = t.2^x + 1$$

If we are using the Goppa code $\Gamma(L, g)$ with codewords of length n, then we want δ_t to be less than n. Hence we can calculate the value of x by the following equation

$$x = \lfloor log_2(\frac{n-1}{t}) \rfloor$$

From the value of x we can calculate the length $l = x.t$. We can see if we use this method then decoding is also very easy.

Remark 5.0.1 *Main draw-back of this encoding system is that maximum value of* δ_t *we can choose is* $t.2^x + 1$. *Also the length(l) of binary message* m *is less than* $log_2 \begin{pmatrix} n \\ t \end{pmatrix}$. *But our encoding method have time complexity linear in* l *hence our method is much faster than the existing constant weight encoding method given by Nicolas Sendrier. We can increase the length* l *of the message* m, *if we have the effective algorithm to calculate the maximum value of* $\sum x_i.l_i$ *less then equal to* $log_2 \begin{pmatrix} n \\ t \end{pmatrix}$, *where* x_i *is the number of subarrays of length* l_i. *We also have to keep in mind that sum of all the subarrays should be exactly equal to* t *and the maximum* δ_t *should be less than* n.
This algorithm is expected to be much faster than existing old methods. This encoding method is easy to implement and lesser complexity.

Appendix A

Pseudo Codes

We have divided all the Pseduo Codes in following five sections:

A.1 Galois Field

Algorithm A.1.1 Pseudo code for cardinality and multiplicative order of extension field

Input: deg.ext=Degree of extension
$gf.cardinality \leftarrow 1 << deg.ext$
$gf.mult.order \leftarrow ((1 << deg.ext) - 1)$

Algorithm A.1.2 Code for exponentiation

Input: deg.ext.
$gf.exp[0] \leftarrow 1$
for $i = 1$ to gf.mult.ord **do**
  $gf.exp[i] \leftarrow gf.exp[i-1] << 1$
end for
if $gf.exp[i-1]\&(1 << deg.ext - 1)$ **then**
  gf.exp[i]$\wedge = prim.poly(deg.ext)$
end if
$gf.exp[gf.mult.order] = 1$

Algorithm A.1.3 Code for logarithm

Input: deg.ext.
gf.log[0]← gf.ord()
for $i = 1$ to gf.mult.ord **do**
  $gf.exp[i] \leftarrow i$
end for

Algorithm A.1.4 Code for modular function

Input: q if $q \geq gf.mult.order$.
return $(q\&gf.mult.order \oplus (q >> deg.ext())$

Algorithm A.1.5 Code for addition(gf.add), multiplication(gf.mult) and division(gf.div)

Input:two element x and y of the field F_{2^m}.
Output: gf.add(x, y)=addition of x and y
return $(x \oplus y)$
Output: gf.mul(x,y)= multiplication of x and y
if $x = 0$ or $y = 0$ **then**
  return 0
else
  return $gf.exp[gf.mod(gf.lox[x] + gf.log[y])]$
end if
Output: gf.div(x,y)=x divided by y
if x=0 **then**
  return 0
else
  return $gf.exp[gf.mod(gf.lox[x] - gf.log[y])]$
end if

Algorithm A.1.6 Code for square(gf.sqr), square root(gf.sqrt) and inverse(gf.inv)

Input:A element x of the field F_{2^m}.
Output: gf.sqr(x)=square of x
if x=0 **then**
return 0
return $gf.exp[gf.mod(gf.lox[x] << 1)]$
end if
Output: gf.sqrt(x)=square root of x
if $x = 0$ **then**
return 0
return $gf.exp[gf.mod(gf.lox[x] << (deg.xt() - 1))]$
end if
Output: gf.inv(x)=inverse of x
return $gf.exp[gf.mult.ord() - gf.log[x]]$

Algorithm A.1.7 Code for calculating exponent

Input:Two elements x and y of the field F_{2^m}.
Output: gf.pow(x, y)=x^y
if x=0 **then**
return 0
else
if y=0 **then**
return 1
end if
else
while y>>deg.ext() **do**
$y \leftarrow$(y & (gf.mult.order())) + (y >> deg.ext())
end while
$y* = gf.log[x]$
while y>>deg.ext() **do**
$y \leftarrow$(y & (gf.mult.order())) + (y >> deg.ext())
end while
return gf.exp[i]
end if

A.2 Polynomial

Algorithm A.2.1 Code for calculating degree of a polynomial

Input:a polynomial p.
Output: degree of the polynomial
d← p.size-1
while (($d \geq 0$)&& (p.coeff(d)==0)) **do**
–d
end while
deg(p)=d
return d

Algorithm A.2.2 Code for multiplying two polynomials

Input:two polynomials p and q.
Output: multiplication of two polynomials
poly.calculate.deg(p)
poly.calculate.deg(q)
dp=deg(p), dq=deg(q)
r=poly.alloc(dp+dq) note:poly.alloc(Allocate a memory for the polynomial of size dp+dq+1)
for $i = 0; i <= dp; ++ i$ **do**
for $j = 0; j <= dq; ++ j$ **do**
$r(i + j) = r(i + j) + p(i) * q(j)$
end for
end for
poly.calculate.deg(q)
return r

Algorithm A.2.3 Code for evaluating a polynomial p(x) at some x=a

Input:a polynomial $p(x)$ and a point $x = a$.
Output: $p(a)$
$d = deg(p)$
$b = p.coeff[d--]$
for $; d >= 0; --d$ **do**
  if $b \neq 0$ **then**
    $b = b * a + p.coeff[d]$
  else
    $b = coeff[d]$
  end if
end for
return b

Algorithm A.2.4 Code for GCD of two polynomials using

Input:Two polynomials p and q, deg(p) and deg(q) are degrees of polynomial p and q respectively.
Output: gcd(p,q)
if $deg(p) \leq deg(q)$ **then**
  $a \leftarrow poly.copy(p)$,$b \leftarrow poly.copy(q)$
  $g1 \leftarrow 1$, $g2 \leftarrow 0$, $h1 \leftarrow 0$, $h2 \leftarrow 1$
  while a$\neq 0$ **do**
    $j \leftarrow deg(a) - deg(b)$
    if $j < 0$ **then**
      $a \leftarrow b$
      $g1 \leftarrow g2$
      $h1 \leftarrow h2$
      $j \leftarrow -j$
    end if
    $u \leftarrow u + x^j v$
    $g_1 \leftarrow u + x^j g_2$
    $h_1 \leftarrow u + x_2^h v$
  end while
  $d \leftarrow v$, $g \leftarrow g_2$, $h \leftarrow h_2$
else
  $gcd(q, p)$
end if
return d, g, h $\{d = g * p + h * q\}$

Algorithm A.2.5 To calculate quotient of a polynomial p with respect to another polynomial q

Input:Two polynomials p and q, deg(p) and deg(q) are degrees of polynomial p and q respectively.

Output: quotient(p,q)

$r \leftarrow p$,$s \leftarrow poly(deg(p) - deg(q))$ Note: poly(k) gives a polynomial structure of size $k + 1$.

deg(s)$\leftarrow$ deg(p)-deg(q)

$a \leftarrow \dfrac{1}{q[deg(q)]}$

for i=deg(p);$i \geq deg(q)$;–i **do**

  b$\leftarrow a * r[i]$

  s[i-deg(q)]$\leftarrow$ b

  if b$\neq$0 **then**

    r[i]$\leftarrow$ 0

  end if

  for j=i-1;j$\geq$ i-deg(q);–j **do**

    r[j]=r[j]+b*q[deg(q)-i+j]

  end for

end for

return s {p=q*s+r}

Algorithm A.2.6 To calculate remainder of a polynomial p with respect to another polynomial q

Input:Two polynomials p and g, deg(p) and deg(g) are degrees of polynomial p and g respectively.
Output: remainder(p,g)
d← $deg(p) - deg(g)$
if d≥ 0 **then**
  a← $\frac{1}{g[deg(g)]}$
  for (i=deg(p);d≥ 0; –i;d++) **do**
    if p[i]≠ 0 **then**
      b← $a * p[i]$
      for (j=0;d<deg(g);++j) **do**
        p[j+d]← p[j+d]+b*g[i]
      end for
      p[i]←0
    end if
  end for
  deg(p)← deg(g)-1
  while ((deg(p)≥ 0)&&(p[deg[p]==0)) **do**
    deg(p)← deg(p)-1
  end while
end if

Algorithm A.2.7 To Generate a irreducible monic polynomial over field F_{q^m}

Input:Size of the polynomial t.
Output: polynomial g
deg(g)←t
g[t]←1
while g is irreducible **do**
  i=0
  for (i=0;i<t;++i) **do**
    g[i]←g[rand(1,n)]
  end for
end while

Algorithm A.2.8 To square a polynomial in field F_{2^m}

Input:a polynomial p and irreducible polynomial g.
Output: polynomial $sq = p^2 mod(g)$
d=deg(g)
for i=0 to $d/2 - 1$ **do**
  $sq[i] = p^{2i}$
end for
for i=d/2 to d-1 **do**
  $sq[i] = p^2.sq[i-1] mod(g)$
end for
adjust coefficient and update degree of sq
return sq

Algorithm A.2.9 To check irreducibility of a polynomial

Input:a polynomial g.
Output: True or false
f(x)← x
for i=1 to $deg(g)/2 * deg.ext()$ **do**
  $f(x) = f(x)^2$ mod(g(x))
end for
if gcd(g(x),f(x)-x)≠1 **then**
  return TRUE
else
  return FALSE
end if

A.3 Matrix

Algorithm A.3.1 Gaussian Elimination

```
Input: Generator matrix H of size r × n
Output: New generator matrix R of size r × (n − r), and a permutation
array P of size n
max←coln-1
for i=1 to n do
  P[i]← i
end for
failcnt←0
for i=1 to rown(H), max– do
  findrow←0
  for j=i to rown(H) do
    if H[j,max] then
      if i≠j then
        H[i, :]∧ = H[j, :]
      end if
      findrow =1
      break
    end if
  end for
  if findrow=0 then
    P[coln[H]-rown[H]-1-failcnt]←max
    failcnt++
    if max=0 then
      return null
    end if
    i–
  else
    P[i+coln[H]-rown[H]←max
    for j=i+1 to rown(H) do
      if H[j,max] then
        H[j, :]∧ = H[i, :]
      end if
    end for
    for j=i-1 to 0, j– do
      if H[j,max] then
        H[j, :]∧ = H[i, :]
      end if
    end for
  end if
end for
R←0
for i=1 to i=n-r do
  for j=0 to j=r do
    if H[j, P[i]]=1 then
      Rchange[i,j]
    end if
```

Algorithm A.3.2 Matrix Multiplication

Input:two Matrix A and B.
Output: Matrix $C = AB$, with dimension (rown(A),coln(B))
if coln(A)neqrown(B) **then**
  C=NILL
else
  for i=1 to i=rown(A) **do**
    for j=1 to j=coln(B) **do**
      for k=1 to k=rown(N) **do**
        C[i,j]=$\sum_{k=1}^{n}$A[i,k]B[k,j]
      end for
    end for
  end for
end if
return C

A.4 Key-Generation

Algorithm A.4.1 Permuting support array L

Input:Support L and permutation array P.
Output: Permuted array L'
if $P = Id$ **then**
  $L' \leftarrow L$
else
  for i=0 to i=1 **do**
    $L'[i] \leftarrow L[P[i]]$
  end for
end if
return L'

Algorithm A.4.2 Key-generation

Input:degree of extension m and size of error t.
Output: Public Key $PK = R$ **and Secret Key** $SK = (L', g)$
L$\leftarrow$ perm(F_{2^m})
$g \leftarrow random.irred.poly(t)$
$(R, P) \leftarrow get.Matrix.Perm(L, g)$
$L' \leftarrow permute(L, P)$
$PK \leftarrow R$
$SK \leftarrow (L', g)$
return PK, SK

A.5 Encryption

Algorithm A.5.1 Encryption

Input: Message m of block length k and public key pk.
Generate a random binary array $e[\,]$ of length n and hamming wait t.
%% now we will calculate mR
for $i = 0$ to $i < n - k$ **do**
  for $j = 0$ to $j < k$ **do**
    $mR_i \rightarrow \sum_{j=0}^{j=k-1} m_j.R_{(j,i)}$
  end for
end for
ciphertext= $[m|mR] + e[\,]$
return ciphertext

A.6 Decryption

Algorithm A.6.1 Syndrome Computation

Input: Receved word $c = \{c_1, c_2, ..., c_n\}$ and collection of $f'_{\alpha_i}s$ where $1 \leq i \leq n$

$R \rightarrow 0$
for from $i = 1$ to $i = n$ **do**
$R \rightarrow R + c_i.f_{\alpha_i}(z)$ mod $(g(z))$
end for
return R

Algorithm A.6.2 Solving Key Equation

Input: Received cipher text y and Goppa Code $\Gamma(L, g)$.
compute syndrome $R_c(z)$ for the cipher text y.
compute $T(z) = 1/(R_c(z) \quad mod(\ \ g(z))$
compute $S(z) = \sqrt{(T(z) + z)}$
Now Solve $\sigma_1(z).S(Z) = \sigma_0(z)$ using Extended Euclidean Algorithm.
compute the error locator polynomial $\sigma(z) = \sigma_1(z)^2 + z.\sigma_1(z)^2$
return $\sigma(z)$

Algorithm A.6.3 Berlekamp Trace Algorithm

Input: Error locator polynomial $\sigma(z)$,$d = degree(\sigma(z))$ and $i = 0$. Function is written as $BTA(\sigma, i, d)$
if degree($\sigma(z)$=0) **then**
  return 0
end if
if degree($\sigma(z) = 1$) **then**
  return $root[i] \leftarrow \sigma(z)_0/\sigma(z)_1$
end if
$gcd_1 \leftarrow gcd(\sigma, Tr(\alpha^i.z)$
$gcd_2 \leftarrow gcd(\sigma, 1 + Tr(\alpha^i.z)$
$e \leftarrow degree(gcd_1)$
return BTA(gcd_1, i+1,e), BTA(gcd_2, i+1,d-e)

Bibliography

[1] *E. R. Berlekamp.* Algebraic Coding Theory. *Aegen Park Press, 1968.*

[2] *E. R. Berlekamp, H. Rumsey, and G. Solomon. On the solution of algebraic equations over finite fields. volume 10, pages 553–564, June 1967.*

[3] *Daniel J. Bernstein, Tanja Lange, and Christiane Peters. Attacking and defending the mceliece cryptosystem. pages 31–46, 2008.*

[4] *Bhaskar Biswas and Nicolas Sendrier. McEliece cryptosystem implementation: Theory and Practice. In* PQCrypto*, pages 47–62, 2008.*

[5] *P. Camion. An iterative euclidean algorithm. Rapport de recherche 844, INRIA, May 1988.*

[6] *W. Diffie and M. Hellman. New directions in cryptography.* IEEE Transactions on Information Theory*, 22(6):644–654, November 1976.*

[7] *Matthieu Finiasz and Nicolas Sendrier. Security bounds for the design of code-based cryptosystems. 5912:88–105, 2009.*

[8] *Philippe Gaborit. Shorter keys for code based cryptography.* International Workshop on Coding and Cryptography - WCC, Bergen, Norway*, 2005.*

[9] *I. Herstein.* Topics in Algebra. *John Wiley, New York, 1975.*

[10] *K. Kobara and H. Imai. Countermeasure against reaction attacks (in japanese). In* The 2000 Symposium on Cryptography and Information Security : A12*, January 2000.*

[11] *R. Lidl and H. Niederreiter.* Finite Fields. *Cambridge University Press, 1983.*

[12] *P. Loidreau and N. Sendrier. Some weak keys in McEliece public-key cryptosystem. In* IEEE Conference, ISIT'98*, Cambridge, MA, USA, August 1998.*

[13] P. Loidreau and N. Sendrier. Weak keys in McEliece public-key cryptosystem. IEEE Transactions on Information Theory*, 47(3):1207–1212, April 2001.*

[14] R. J. McEliece. A public-key cryptosystem based on algebraic coding theory. DSN Prog. Rep., Jet Prop. Lab., California Inst. Technol., Pasadena, CA*, pages 114–116, January 1978.*

[15] H. Niederreiter. Knapsack-type crytosystems and algebraic coding theory. Prob. Contr. Inform. Theory*, 15(2):157–166, 1986.*

[16] N. J. Patterson. The algebraic decoding of Goppa codes. IEEE Transactions on Information Theory*, 21(2):203–207, March 1975.*

[17] N. Sendrier. Encoding information into constant weight words. In IEEE Conference, ISIT'2005*, Adelaide, Australia, September 2005.*

[18] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM J. Comput.*, 26(5):1484–1509, 1997.*

[19] V. M. Sidel'nikov and S. O. Shestakov. On cryptosystem based on generalized Reed-Solomon codes. Discrete mathematics (in russian)*, 4(3):57–63, 1992.*

[20] V.A. Zinoviev. On the solution of equations of degree ≤ 10 *over finite fields GF(*2^q*). 1996.*