An Programmatic Introduction to the McEliece Cryptosystem

Daniel Medina, Yinwei (Charlie) Zhang

March 23, 2017

Contents

Abstract

In this paper, we examine the McEliece cryptosystem, published by McEliece in 1978. We explore the details behind the system and examine binary Goppa codes. We then look at the different parameters of the system. Finally, we provide an implementation of the McEliece cryptosystem in Sage, a Python based language.

1 Introduction

Cryptography lets us communicate securely. Cryptosystems are used for exchanging messages, for identification with signatures, and even in currencies. Many modern cryptosystems are public key based, since then the channel of communication, like the internet, doesn't have to be secure. Other people can grab the encrypted information, but can't decrypt it in a reasonable amount of time. Public key cryptosystems are built on asymmetric decoding times, so the person with the private key can decrypt it fast, in P time, while an attacker can only decrypt it slowly, in NP time.

Modern cryptosystems, like the popular RSA cryptosystem (1), are based on factoring large integers into primes. Factoring primes is in NP, since we can verify that $p \mid n$, but we currently can't solve the problem in polynomial time. However, with the introduction of quantum computers, it turns out that we can factor numbers in polynomial time. Shor found such a quantum algorithm in 1994 (5). Therefore, the attacker can decrypt a encrypted message in a reasonable amount of time without knowing the private key, breaking the cryptosystem.

So in a post-quantum world (2), cryptosystems that depend on prime factorization, like RSA, elliptic curves, don't work. There are some other NP problems in number theory that don't have polynomial time solvability in quantum computers, and there are cryptosystems based on those aren't broken. The McEliece cryptosystem uses coding theory, particularly linear codes, and the Goppa code as the NP problem. It currently has no known polynomial time algorithm.

2 McEliece

The McEliece cryptosystem (3) is an asymmetric public key cryptographic system that uses error correcting codes for generating the public and private keys, and consequently, for the encryption and decryption processes. The McEliece cryptosystem makes use of binary Goppa codes in its original description. Note that the generation of the keys and the encryption process itself is probablistic. Both processes choose randomly a Goppa code and random vector respectively. The decryption process is deterministic.

2.1 Parameters

The alphabet of the cryptosystem is $\{0, 1\}^n$, where n is the length of the code, so the alphabet is binary. The other parameter is t, which is the maximum number of errors to be corrected by internal code.

2.2 Key Generation

G' is the scrambled code matrix, and is defined as

$$G' = SGP$$

S is a scramble matrix, a non-invertible matrix of dimensions (k, k). G is the generator matrix from our linear code of dimensions (k, n). P is a permutation matrix of dimensions (k, k). A permutation matrix The resultant matrix, G' has dimensions (k, n).

The public key is (G', t).

The private key is (S, G, P). If we are using Goppa codes, we also need the Goppa polynomial used for generation in order to efficiently decode the Goppa portion.

2.3 Encryption

Then, the encryption scheme can be described as follows: Alice publishes her public key. Bob wants to encrypt and send a message m. Bob encrypts the message by first choosing a random vector z of length k and of weight t. The message m is encoded using the matrix G' found in the public key and the vector z is added to the encoded message. The ciphertext is thus given by

$$C = mG' + z$$

2.4 Decryption

Alice receives the ciphertext C and proceeds to decrypt using her private key.

By calculating CP^{-1} Alice gets $(mS)G + zP^{-1}$. If CP^-1 is decoded using the decoding algorithm for the chosen Goppa code, then mSG will be obtained. The message m can be found by multiplying with the inverses of S and $G : m = mSG(S^-1)G^{-1}$.

3 Goppa Codes

In his paper in 1978, McEliece recommended the linear code that should be used, G, be a binary Goppa Code.

3.1 Parameters

Let $GF(p^m)$ be a Galois Field, p be prime, and m be an integer.

A Goppa code is a $[n, k, d]$ linear code, where n is the code length, k is the length of the message itself, and d is the minimum distance between each codeword. To correct t errors, we choose a Goppa polynomial of degree t. For Goppa codes, $n = 2^m$. k is the dimension of the Goppa polynomial and $k \geq n - mt$. The Hamming distance is $d \geq 2t + 1$ (4).

3.2 General Goppa Codes.

3.2.1 Creating a Goppa Code

A Goppa code Γ is made from two components, L and g. That is

$$\Gamma = (L, g)$$

L is a set of elements that represents $GF(p^m)$ and is written as $GF(2)\alpha.Then$

$$L = \{\alpha_1, \alpha_2, \ldots, \alpha_n, L \in GF(p^m)$$

g is the Goppa polynomial. It's an irreducible polynomial in $GF(p^m)$ with degree t, so we choose the Goppa polynomal to correct up to t errors. Then

$$g = g_0 + g_1(x) + \cdots + g_t(x^t) = \sum_{i=0}^{t} g_i x^i$$

For the Goppa code to be valid, $g(L) \neq 0$. That is, each $g(\alpha_i) \neq 0$ for each $alpha_i \in L$.

3.2.2 Generating valid code words

To generate a valid code c, in C, the valid set of codewords in Goppa code Γ, we define

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

A codeword c is valid if and only if $R_c(x) \equiv 0 \pmod{g}$. That means Goppa polynomial $g(x)$ divides $R_c(x)$.

3.3 Binary Goppa Codes.

With binary Goppa codes, our Galois Field is $GF(2^m)$. Note with binary Goppa codes, we can correct up to t errors, with t being the degree of the Goppa polynomial. For Galois Fields with p in $GF(p^m)$, $p > 2$, that maximum number of correctable errors is $\frac{t}{2}$. (4)

3.3.1 Finding the parity check matrix

For binary Goppa codes, the parity check matrix H is

$$\mathrm{H} = \mathrm{VD} = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ L_0^1 & L_1^1 & L_2^1 & \cdots & L_{n-1}^1 \\ L_0^2 & L_1^2 & L_2^2 & \cdots & L_{n-1}^2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ L_0^t & L_1^t & L_2^t & \cdots & L_{n-1}^t \end{pmatrix} \begin{pmatrix} \frac{1}{g(L_0)} & & & & \\ & \frac{1}{g(L_1)} & & & \\ & & \frac{1}{g(L_2)} & & \\ & & & \ddots & \\ & & & & \frac{1}{g(L_{n-1})} \end{pmatrix}$$

where V is the Vandermonde matrix with α from L, and D is the Identity matrix weighted with $g(L_i)$.

3.3.2 Finding the generator matrix

The generator matrix, by definition, is the null space matrix of H. That is

$$GH^T = 0$$

We can use that to solve for G.

3.3.3 Encoding Binary Goppa Codes

Encoding a message with is simple. We find

$$c = \mu G$$

where c is the encrypted ciphertext, μ is the original message, and G is the generator matrix.

3.3.4 Decoding Binary Goppa Codes

If the recieved encrypted text c has no errors, then $Hc^T \equiv 0 \pmod 2$.

To decode a an error free message (4), we find

$$c = \mu G$$
$$G^T \mu = c$$

We can find μ by finding $(G^T \mid \mu)$ and use to reduction to solve it so that

$$\mathrm{G^T\ c} = \begin{pmatrix} 1 & 0 & \cdots & 0 & \mu_1 \\ 0 & 1 & \cdots & 0 & \mu_2 \\ \cdots & \cdots & \ddots & \cdots & \cdots \\ 0 & 0 & \cdots & 1 & \mu_k \\ \hline & & X & & \end{pmatrix}$$

3.3.5 Correcting Errors in Codewords

If the recieved encrypted text c has errors, then $Hc^T \not\equiv 0 \pmod 2$.

Let $y = (y_1, y_2, \ldots, y_n)$ be the recieved encrypted message with $r \leq t$ errors. Then

$$y = c + e$$

where c is the ciphertext, and e is the error vector. Note that $e = 0$ when there are no errors, and the maximum Hamming weight is for e is t. We need to find the elements in e where $e_i = 1$.

To do that, in $GF(2^m)$, let's define the error locating polynomial σ. Let E be the set of locations where $e_i = 1$. Then

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

To find the error locating polynomial, we can use Patterson's algorithm (6). There are alternatives, such as Berlekamp's algorithm, but Patterson corrects up to t errors, while Berlekamp corrects only up to $\frac{t}{2}$.

Note that since our code is binary, we just need to locate the
First compute the syndrome polynomial, s, which is

$$s(x) = \sum_{i=1}^{n} y_i(x - \alpha_i) \bmod g(x)$$

where α is from L, and $g(x)$ is the Goppa polynomial.
Then we find σ. We find $h(x)$, the inverse of the syndrome, so

$$s(x)h(x) \equiv 1 \bmod g(x)$$

If $h(x) = x$, then $\sigma(x) = x$.
Otherwise, find $d(x)$, so that $d^2(x) \equiv h(x) + x \bmod g(x)$. Then find $a(x), b(x)$ so that

$$d(x)b(x) \equiv a(x) \bmod g(x)$$

We minimize $b(x)$ to have the lowest degree possible. Then

$$\sigma(x) = (b^2(x))x + a^2(x)$$

Afterwards, we use σ to get E, the locations in error vector e where $e_i = 1$. Then we solve

$$c = y - e$$

Then we've corrected the errors, and can decode it now.

4 Analysis of the McEliece cryptosystem

4.1 Why Goppa codes?

Goppa codes have an efficient polynomial time decoding algorithms (7), so decryption is fast.

There are also a relatively large density of Goppa codes. For a given code with length n and degree t of the Goppa polynomial, there are no known number of Goppa polynomials. Ryan and Fitzpatrick found a upper bound, and the bound grows exponentially as n increases and as t increases (8). So for a brute force attack, the number of codes increases exponentially, so finding the right code is a NP problem.

4.2 Why binary Goppa codes?

Binary Goppa codes lets us decode up to t errors, with t refering to the degree of the Goppa polynomial, while other Goppa codes only lets us decode $\frac{t}{2}$. Therefore, binary Goppa codes have a better bitrate, and are more efficient. Binary Goppa codes meet the Gilbert-Varshamov bound as $n \to \infty$.

4.3 Drawbacks

The biggest disadvantage of the McEliece cryptosystem is the size of the public key. Recall that the public key is (G', t), and G is of dimensions (k, n). So to represent the matrix, we would need kn bits. McEliece originally called for $n = 1024, 524$, so our size is 524KB, which is huge.

4.4 Attacks

Modern attacks not the linear code G used in the McEliece system, but from Information Set Decoding (ISD) (11). In 2012, the fastest method found for a (1024, 524, 50) Goppa code (specified by McEliece) takes $2^{59.9}$ work cycles. With 200 cores, one could solve this in about 7 hours. To combat this, we need to pick (n, k) to be bigger. See (11) for more info.

5 Programming the McEliece cryptosystem

5.1 Overview

Implementing the McEliece cryptosystem was not a trivial task. Several difficulties were presented during the process due to its abstract algebra characteristics. Specifically, the part that gave us more problems was to implement the Goppa Code class. Initially, we tried implementing everything from scratch in pure Python. However, we soon realized that this would take a tremendous amount of time as there are many building blocks that need to be implemented first. For example, before even implementing Goppa codes, we needed first to implement Galois Fields, which would also require us to implement a comprehensive class for fields, polynomials, and matrices including implementing functions for most of their operations.

For this reason we decided to look for libraries that we could use to built our McEliece and Goppa code implementations on top of them. We first found a Gallois Fields library that also included some functions for matrices operations and polynomials. However, we found limitations that did not allowed us to fully implement the Goppa code. For this reason we had to discard the work we already had done with that library and looked for a more powerful one. Fortunately, we found SageMath which is a very powerful Math toolset that is based on math libraries for Python. It has complete libraries for matrices, polynomial rings, and even Galois Fields. Using this library and basing some parts of our code in this paper and example found:[?], [?] we were able to finish our McEliece implementation.

5.2 Code Walkthough

Our code is comprised of mainly two classes. The McEliecePKS class and the BinGoppaCode class. The BinGoppaCode class constructor takes two parameters which are m and t and are used to generate the n and k parameters of the Goppa Code.

```
def __init__(self, m, t):
  self.m = m
  self.t = t
  self.n = 2**m
  self.k = self.n-m*t"
```

Then, the Galois field objects are generated using the GF class of SageMath.Having the extension Galois field then it is easy to define the subset L of alpha roots.

A SageMath polynomial ring object under the Galois field extension is created to serve as the Goppa polynomial. For easier implementation, its defined to be of the form $x^t + (\alpha_i)x + \alpha_i$

If the proposed polynomial fails to be irreducible or either has an α from L as root, then the next polynomial with coefficients α_{i+1} is tried.

```
#Generating galois field
gf2.<a> = GF(2)
gf2m.<a> = GF(2**self.m)
self.a = a
self.gf2 = gf2
self.gf2m = gf2m

#Choosing n elements for L, choosing from power 2 to power 2+n
L = [a**i for i in range(2,2+self.n)]
self.L = L

#Generating goppa polynomial
```

```
  foundPoly = False
  x = PolynomialRing(gf2m,repr(a)).gen()
  for i in range(len(L)):
    gpoly = x^t + L[i]*x + L[i]
    #Checking that satisfies Goppa polynomial conditions e.g.Irreducible and elements o
    foundRoot = False
    for elem in L:
      if gpoy(elem) == gf2m(0):
        foundRoot = True
        break
      #If poly is irreducible and non rooots where found then finsh, otherwise try anot
      if gpoly.is_irreducible() and not foundRoot:
        foundPoly = True
        break

  if not foundPoly:
    print "Could not generate Goppa polynomial"
    return

  self.gpoly = gpoly
```

The last part of the BinGoppaCode class constructor consists of generating the parity matrix and the generator matrix.The parity matrix is first constructing using the Goppa polynomial and the alphas from the subset L following the definition of the matrix H. Then, the matrix G can easily be generated as G is formed by the vectors of the nullspace of H.This section was built with the help of the paper [?]

```
  self.H_gRS = matrix([[L[j]^(i) for j in range(self.n)] for i in range(self.t)])
  self.H_gRS = self.H_gRS*diagonal_matrix([1/gpoly(L[i]) for i in range(self.n)])
  self.H_Goppa = matrix(gf2,self.m*self.H_gRS.nrows(),self.H_gRS.ncols())

  for i in range(self.H_gRS.nrows()):
    for j in range(self.H_gRS.ncols()):
      be = bin(self.H_gRS[i,j].integer_representation())[2:];
      be = be[::-1];
      be = be+'0'*(m-len(be));
      be = list(be);
      self.H_Goppa[m*i:m*(i+1),j] = vector(map(int,be));
      self.G_Goppa = self.H_Goppa.transpose().kernel().basis_matrix();
      G_Goppa_poly = self.H_gRS.transpose().kernel().basis_matrix();
```

The two main methods from the class BinGoppaCode are encode(u) and decode(y). The encode method is very simple as we already have built the generator matrix:

```
def encode(self, u):
  return u*self.G_Goppa
```

The decode method is more complex. There are many decoding algorithms but we decided to use the Patterson's algorithm due to being more simple when using Binary Goppa Codes. The Patterson's algorithm is explained at the previous Goppa Code section and the implementation uses the libraries and is based on the one found at (10).

```
synd = self.SyndromeCalculator*y.transpose();

syndrome_poly = 0;
for i in range (synd.nrows()):
  syndrome_poly += synd[i,0]*X^i

error = matrix(GF(2),1,self.H_Goppa.ncols());
(g0,g1) = self._split(self.gpoly);
(d,u,v) = xgcd(g1,self.gpoly)
g1_inverse = u.mod(self.gpoly)
sqrt_X = g0*g1_inverse;
T = syndrome_poly.inverse_mod(self.gpoly);
(T0,T1) = self._split(T - X);
R = (T0+ sqrt_X*T1).mod(self.gpoly);
#Perform lattice basis reduction.
(alpha, beta) = self._lattice_basis_reduce(R);
#Construct the error-locator polynomial.
sigma = (alpha*alpha) + (beta*beta)*X;
#Pre-test sigma if fails, then zerro error vector is returned
if (X^(2^self.m)).mod(sigma) != X:
  return y+error;
#Generating the error correcting vector
  for i in range(len(self.L)):
    if sigma(self.L[i]) == 0:
      error[0,i] = 1;

return y+error;
```

The McEliecePKS class is comprised of the encrypt, decrypt, and constructor methods. The constructor takes the arguments m and t very similarly to the BinGoppaCode constructor but additionally besides generating n and k it will have a public key, private key, and the BinGoppaCode object as members.

```
self.m = m
self.t = t
self.n = 2**m
self.k = self.n-m*t
self.pub_key = None
self.priv_key = None
self.goppacode = BinGoppaCode(m,t)
```

The first thing it does is to generate a kxk random scramble matrix S. Although it is randomly generated, it is very likely that the first try will be invertible. The nxn permutation matrix P is also randomly generated, and has only a single 1 per row and column. Now we can generate the public and private keys. The public key is formed by the pair $(S*G*P, t)$ and the private key consists of $(GPoly, G, S, P)$

```
## Generating scramble matrix S
while True:
  S = matrix(gf2, [[randint(0,1) for i in range(self.goppacode.G_Goppa.nrows())]for i i
  if S.is_invertible():
    break

## Generating permutation matrix P
rng = range(n); P = matrix(GF(2),n);
for i in range(n):
  p = floor(len(rng)*random());
  P[i,rng[p]] = 1; rng=rng[:p]+rng[p+1:];

G_ =   S*self.goppacode.G_Goppa*P
self.pub_key = (G_, self.t)
self.priv_key = (self.goppacode.gpoly, self.goppacode.G_Goppa, S, P)
```

The encrypt function encodes the message using the public key $S * G * P$ matrix. Then, it generates an error vector with weight equal to t and adds it to the encoded message which is returned as the encrypted message.

```
def encrypt(self, m, pub_key):
  gf2 = GF(2)
  G_ = pub_key[0]
  t = pub_key[1]
  c = m*G_

  # random error vector
  e = [0 for i in range(G_.ncols())]
  for i in range(t):
    e[randint(0,G_.ncols()-1)] = 1
    e = matrix(gf2, e)
    c = c + e
    return c
```

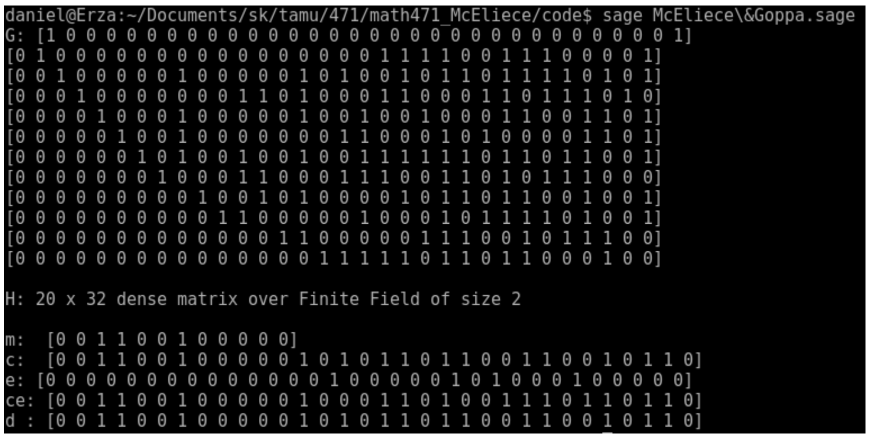
The matrix operations already implemented in the SageMath library makes the decrypting function very simple to implement. First, the inverse of P is calculated and multiplied to the encrypted message. The resulting product is decoded using the BinGoppaCode decoding function and finally a system of equations is solved to get the plaintext message. The commented code lines are how the code would have looked liked if not using the SageMath library.

```
def decrypt(self, c):
  priv_key = self.priv_key
  P = priv_key[3]
  S = priv_key[2]
  gpoly = priv_key[0]
  G_Goppa = priv_key[1]

  #Getting inverse of P
  #m = P.join_with(Matrix.get_identity(P.rows))
  #m = m.get_reduced_echelon()
  #m = m.submatrix(P.rows,P.rows,m.rows,m.cols)
  #m = m*c
  m = c*(~P)
```

```
    # Correcting errors - Solving system
    m = self.goppacode.decode(m)
    #m = (gpoly.transpose()).join_with(m.transpose())
    #m = m.get_reduced_echelon()
    ## finding inverse of S
    #s_ = S.join_with(Matrix.get_identity(S.rows))
    #s_ = m.get_reduced_echelon()
    #s_ = m.submatrix(S.rows,S.rows,m.rows,m.cols)
    #m = s_*m
    m = (S*G_Goppa).solve_left(m)

    return m
```

5.3 Tests

BinGoppaCode McEliecePKS running demonstrationsvusing $m = 5$, $t = 4$, $n = 2$, $k = n - mt$, and random messages m.

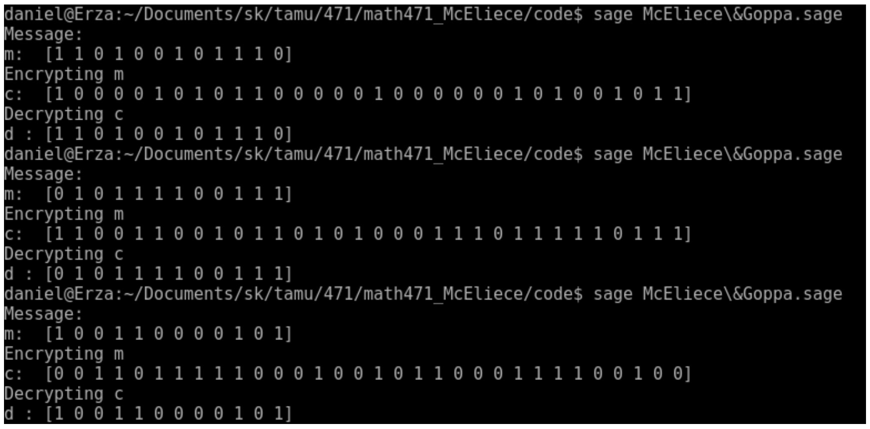


```
daniel@Erza:~/Documents/sk/tamu/471/math471_McEliece/code$ sage McEliece\&Goppa.sage
G: [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
[0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 1 1 1 0 0 0 0 1]
[0 0 1 0 0 0 0 0 1 0 0 0 0 0 1 0 1 0 0 1 0 1 1 0 1 1 1 1 0 1 0 1]
[0 0 0 1 0 0 0 0 0 0 0 1 1 0 1 0 0 0 1 1 0 0 0 1 1 0 1 1 1 0 1 0]
[0 0 0 0 1 0 0 0 1 0 0 0 0 0 1 0 0 1 0 0 1 0 0 0 1 1 0 0 1 1 0 1]
[0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 1 1 0 0 0 1 0 1 0 0 0 0 1 1 0 1]
[0 0 0 0 0 0 1 0 1 0 0 1 0 0 1 0 0 1 1 1 1 1 1 0 1 1 0 1 1 0 0 1]
[0 0 0 0 0 0 0 1 0 0 0 1 1 0 0 0 1 1 1 0 0 1 1 0 1 0 1 1 1 0 0 0]
[0 0 0 0 0 0 0 0 0 1 0 0 1 0 1 0 0 0 0 1 0 1 1 0 1 1 0 0 1 0 0 1]
[0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 1 0 0 0 1 0 1 1 1 1 0 1 0 0 1]
[0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 1 1 1 0 0 1 0 1 1 1 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 0 1 1 0 1 1 0 0 0 1 0 0]

H: 20 x 32 dense matrix over Finite Field of size 2

m:  [0 0 1 1 0 0 1 0 0 0 0 0]
c:  [0 0 1 1 0 0 1 0 0 0 0 0 1 0 1 0 1 1 0 1 1 0 0 1 1 0 0 1 0 1 1 0]
e: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 1 0 0 0 1 0 0 0 0 0]
ce: [0 0 1 1 0 0 1 0 0 0 0 0 1 0 0 0 1 1 0 1 0 0 1 1 1 0 1 1 0 1 1 0]
d : [0 0 1 1 0 0 1 0 0 0 0 0 1 0 1 0 1 1 0 1 1 0 0 1 1 0 0 1 0 1 1 0]
```

Figure 1: Goppa

Some testing of increasing m and thus n against running time showed that the running time increased exponentially. Therefore, it is very costly to increase n and the security of the McEliece cryptosystem.

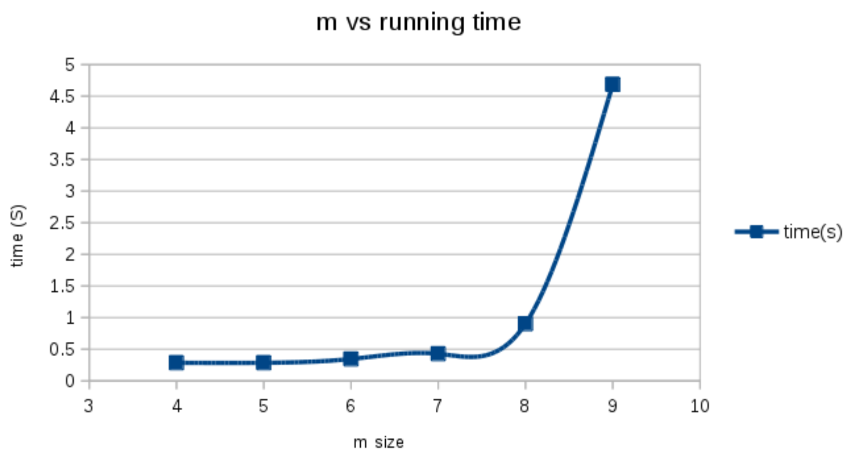
m	time(s)
4	0.2855679989
5	0.2855679989



```
daniel@Erza:~/Documents/sk/tamu/471/math471_McEliece/code$ sage McEliece\&Goppa.sage
Message:
m:  [1 1 0 1 0 0 1 0 1 1 1 0]
Encrypting m
c:  [1 0 0 0 0 1 0 1 0 1 1 0 0 0 0 0 1 0 0 0 0 0 0 1 0 1 0 0 1 0 1 1]
Decrypting c
d : [1 1 0 1 0 0 1 0 1 1 1 0]
daniel@Erza:~/Documents/sk/tamu/471/math471_McEliece/code$ sage McEliece\&Goppa.sage
Message:
m:  [0 1 0 1 1 1 1 0 0 1 1 1]
Encrypting m
c:  [1 1 0 0 1 1 0 0 1 0 1 1 0 1 0 1 0 0 0 1 1 1 0 1 1 1 1 1 0 1 1 1]
Decrypting c
d : [0 1 0 1 1 1 1 0 0 1 1 1]
daniel@Erza:~/Documents/sk/tamu/471/math471_McEliece/code$ sage McEliece\&Goppa.sage
Message:
m:  [1 0 0 1 1 0 0 0 0 1 0 1]
Encrypting m
c:  [0 0 1 1 0 1 1 1 1 1 0 0 0 1 0 0 1 0 1 1 0 0 0 1 1 1 1 0 0 1 0 0]
Decrypting c
d : [1 0 0 1 1 0 0 0 0 1 0 1]
```

Figure 2: McEliece

6	0.3452010155
7	0.4280600548
8	0.9029610157
9	4.6846778393



6 References

1. Rivest, R.; Shamir, A.; Adleman, L. (February 1978). “A Method for Obtaining Digital Signatures and Public-Key Cryptosystems”. Communications of the ACM. 21 (2): 120–126. `10.1145/359340.35934`

2. Daniel J. Bernstein, Johannes Buchmann, Erik Dahmen (editors). Post-quantum cryptography. Springer, Berlin, 2009. Chapter 1. ISBN 978-3-540-88701-0

3. McEliece, Robert J. (1978). "A Public-Key Cryptosystem Based On Algebraic Coding Theory". DSN Progress Report. 44: 114–116. Bibcode:1978DSNPR..44..114M

4. Valentijn, Ashley, "Goppa Codes and Their Use in the McEliece Cryptosystems" (2015). Syracuse University Honors Program Capstone Projects. 845. `http://surface.syr.edu/honors_capstone/845`

5. Shor, Peter W. (1997), "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer", SIAM J. Comput., 26 (5): 1484–1509, arXiv:quant-ph/9508027v2, Bibcode:1999SIAMR..41..303S, `10.1137/S0036144598347011`

6. Daniel J. Bernstein. "List decoding for binary Goppa codes." `http://cr.yp.to/codes/goppalist-20110303.pdf`

7. D. Sarwate, "On the complexity of decoding Goppa codes (Corresp.)," in IEEE Transactions on Information Theory, vol. 23, no. 4, pp. 515-516, Jul 1977. doi: 10.1109/TIT.1977.1055732

8. Fitzpatrick, P.; Ryan, J. (March 2005) "Enumeration of inequivalent irreducible Goppa codes". Discrete Applied Mathematics. 154.2:399-412. `http://dx.doi.org/10.1016/j.dam.2005.03.017`

9. `http://sagemath.org`

10. Risse, T. "How SAGE helps to implement Goppa Codes and McEliece PKCSs". `http://www.ubicc.org/files/zipped/SAGE_Goppa_McEliece_595.pdf`

11. Niebuhr, R., Meziani, M., Bulygin, S. et al. "Selecting parameters for secure McEliece-based cryptosystems" Int. J. Inf. Secur. (2012) 11: 137. `10.1007/s10207-011-0153-2`