Implementation of the McEliece Cryptosystem in Julia*

Ivan Alejandro Moreno Soto

University of North Texas Summer Research Program,
July 23, 2019

1 Post-quantum cryptography

The standard cryptosystems used today depend on the difficulty for computers to solve a specific set of problems, such as integer factorization and computation of the discrete logarithm, but with quantum computers being closer to production every year, there is a serious threat to this security.

Quantum computers will be able to efficiently solve some of the problems that traditional computers cannot, and in turn, will be able to break the security that we use today. To be prepared for the day that quantum computers are a practical reality, we need to develop cryptographic systems that will survive in a post-quantum world.

To accomplish this task, it is necessary to look at other computational problems that are hard for computers to solve, whether they are traditional or quantum. The National Institute of Standards and Technology (NIST) of the United States of America is currently running a competition to standardize a post-quantum public-key encryption scheme. One of the candidates in this competition is the McEliece cryptosystem [8]. This system is based on the general decoding problem: given a received word, get the closest codeword to it in an arbitrary code.

*This work was done when the author visited the Department of Computer Science and Engineering, University of North Texas, USA. The author would like to acknowledge a support from the Universidad de Sonora and the Instituto de Becas y Crédito Educativo del Estado de Sonora.

2 Mathematical background

Let us first review some concepts from abstract algebra and coding theory, which will be useful for the presentation of the McEliece cryptosystem. The coding theory provides a way to add redundant information to a message we want to transmit so that we will be able to detect if errors occurred during communication. Depending on the code, we can even correct up to a certain amount of errors without the need to resend the original message.

The McEliece cryptosystem uses a linear code over a finite field as its primary building block. An (n, k)-code C over a finite field $\mathbb{F}$ is a k- dimensional linear subspace of the vector space $\mathbb{F}^n$. The code C has a generator matrix G whose rows span C over $\mathbb{F}$, and a parity check matrix H that is the right kernel of G. Because computers work in binary, the finite field commonly selected for the code is $GF(2)$, and so we have binary linear codes.

We call the Hamming distance of a vector x to the zero vector the weight of x. The minimum distance of a code is the minimum Hamming distance between any two vectors of the code.

Once we have information we wish to transmit, we can encode it using the generator matrix for the code C. The receiver can then apply a decoding algorithm to detect and correct errors on the message they received.

We can also say that two (n, k) codes C and C' over $\mathbb{F}$ are permutation equivalent if there exists a permutation π such that

$$C' = \pi(C) = \{(x_{\pi^{-1}(1)}, \ldots, x_{\pi^{-1}(n)}) | x \in C\}.$$

There are a few options to choose from to make a linear code, but the McEliece cryptosystem implemented uses binary irreducible Goppa codes.

2.1 Finite fields

Finite fields are those with a finite order. The order has to be a prime number, let us denote it as p, or a prime power p^m. When we use a field with order p^m, its elements are represented as polynomials of degree up to $p^m - 1$ with coefficients in $GF(p)$.

If we set $p = 2$, the field $GF(2^m)$ will be the polynomials of degree up to $2^m - 1$ with binary coefficients. $GF(2^m)$ has characteristic 2, just like $GF(2)$. Addition and substraction are then the same, and we can do it by adding the polynomials term by term. To perform multiplication, we can do classic polynomial multiplication, and then reduce the product modulo an

irreducible polynomial of degree 2^m. Division is done by multiplying the numerator polynomial by the inverse of the denominator modulo an irreducible polynomial. An irreducible polynomial $g(X)$ over $GF(2)$ is a polynomial that cannot be factored into non-trivial polynomials over the same field.

There are multiple irreducible polynomials that we can use to generate $GF(2^m)$, but it does not matter which one we choose, because the fields that they generate are isomorphic.

2.2 Polynomial rings over $GF(2^m)$

To create a Goppa code, we need to find a polynomial with specific properties in the polynomial ring $GF(2^m)[X]$. We can define $GF(2^m)[X]$ as the set of polynomials of the form $P = a_0 + a_1X + \dots a_nX^n$ where every $a_i \in GF(2^m)$. Note that this ring also has characteristic 2.

To successfully create a Goppa code useful for encryption and decryption, we need to get a polynomial that is irreducible over $GF(2^m)$. Having this irreducible polynomial will allow us to compute the inverse $\mod g(X)$ of any polynomial, and thus, create another field that we denote $GF(2^m)[X]/g(X)$.

2.3 Goppa codes

Goppa codes are a type of linear codes that depend on an irreducible monic polynomial over $GF(2^m)[X]$ which we call the Goppa polynomial. Let $m, t \in \mathbb{N}$, the Goppa polynomial has the form:

$$g(X) = \sum_{i=0}^{t} g_i X^i.$$

They also depend on a set $L = \{\gamma_0, \dots, \gamma_{n-1}\}$ of n distinct non-root elements. We call L the code support. With L, we can define the syndrome of a vector $c \in GF(2)^n$ as follows:

$$S_c(X) = -\sum_{i=0}^{n-1} \frac{c_i}{g(\gamma_i)} \frac{g(X) - g(\gamma_i)}{X - \gamma_i} \mod g(X).$$

Alternatively, we can also define a simpler syndrome:

$$S_c(X) = \sum_{i=0}^{n-1} \frac{c_i}{X - \gamma_i} \mod g(X).$$

The binary Goppa code $\mathcal{G}(L, g(X))$ over $GF(2)$ is the set of all $c \in GF(2)^n$ such that the identity $S_c(X) = 0$ holds. Because $GF(2)^n$ has characteristic 2, we simply compute $(X + \gamma_i)^{-1} \mod g(X)$ for every c_i that is not zero.

For every Goppa code, there is a parity check matrix H that we can use to verify if a given vector c belongs to the code. This way, we know that $c \in \mathcal{G}(L, g(X))$ iff $Hc^T = 0$. The matrix H has dimension $t \times n$ and we can transform every polynomial to its binary representation to get a binary matrix of dimension $mt \times n$. To compute this parity check matrix we use matrices X and Y such that $H = XY$, and they are defined as follows:

$$X_{t\times n} = \begin{bmatrix} 1 & 1 & \cdots & 1 \\ \gamma_0 & \gamma_1 & \cdots & \gamma_{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ \gamma_0^{t-1} & \gamma_1^{t-1} & \cdots & \gamma_{n-1}^{t-1} \end{bmatrix},$$

$$Y_{n\times n} = \begin{bmatrix} g(\gamma_0)^{-1} & & & \\ & g(\gamma_1)^{-1} & & \\ & & \ddots & \\ & & & g(\gamma_{n-1})^{-1} \end{bmatrix}.$$

We can compute the generator matrix G by getting the basis of the nullspace of H. The rows of G span $\mathcal{G}$ and it has dimension $k \times n$, with $k \geq n - mt$. With this, we can define a (n, k) Goppa code with length n and dimension k.

The (n, k) Goppa code can detect up to t errors and has minimum distance $2t + 1$. To encode a vector $m \in GF(2)^k$, we compute $c = mG$. To decode a vector $c \in GF(2)^n$, assuming we have at most t errors, we use Patterson's algorithm to find the positions of those errors and retrieve the original codeword. Finally, we solve a linear system of equations, to obtain the plaintext from the corresponding codeword.

Patterson's algorithm takes advantage of the fact that $S_c = S_e \neq 0$. It also defines an error locator polynomial as follows:

$$\sigma_c(X) = \prod_{j \in T_c} (X - \gamma_j),$$

where T_c are the positions of the errors in c. Using the syndrome of c, this algorithm computes σ with the equation

$$\sigma_e(X)S_e(X) = \sigma'_e(X) \mod g(X).$$

After computing the syndrome of the received vector, we need to compute its inverse $T(X) = S_c^{-1} \mod g(X)$. Then, we compute $\tau = \sqrt{T(X) + X}$, and we solve the equation $\beta(X)\tau(X) = \alpha(X) \mod g(X)$ using the extended Euclidean algorithm to get $\beta(X)$ and $\alpha(X)$ of the least degree. We can now get $\sigma_e(X)$ as follows:

$$\sigma_e(X) = c^2 \left((\alpha_i(X))^2 + X(\beta_i(X))^2\right),$$

where $c^2 \in GF(2^m)$ is such that $\sigma_e(X)$ is a monic polynomial. With the error locator polynomial, we can learn the positions in c that have an error by finding its roots.

3 McEliece cryptosystem

The McEliece public-key cryptosystem uses a random-looking binary linear code to encrypt messages. This code comes from a (n, k) irreducible binary Goppa code, find its codeword corresponding to the plaintext, and add a random noise to it. To accomplish this, we compute the generator matrix G' of some Goppa code. Then, we randomly select a binary non-singular matrix $S_{k\times k}$, and a permutation matrix $P_{n\times n}$. We get a random- looking code by constructing a permutation equivalent code as follows:

$$G = SG'P.$$

The public key becomes the tuple (G, t), which allows anybody to encode a message using the permutation equivalent code, and then encrypt it by adding t random errors to the obtained codeword. The private key is $(S, \mathcal{D}_\mathcal{G}, P)$, which allows us to efficiently correct errors using algorithm $\mathcal{D}_\mathcal{G}$, and then to compute the plaintext by reversing the linear transformations done by S and P to the original Goppa code.

This cryptosystem can encrypt and decrypt messages fairly quickly, but at the cost of having somewhat large keys. For details on the McEliece cryptosystem, see [4, 9, 7].

4 Implementation in Julia

An implementation was made in the Julia programming language. It consists of modules for operations in extension fields and polynomial rings, generation of Goppa codes and keys, encryption and decryption of strings.

Operations in extension fields are performed on 32 bits unsigned integers using bitwise operations, using the binary representation of the integer as a way to store a polynomial. Polynomials over $GF(2^m)$ are represented as an array of their coefficients and operations are performed using classic addition, multiplication, and long division. Evaluation is done using Horner's rule to make it faster.

4.1 Key generation

To generate an irreducible binary Goppa code, the program takes a primitive polynomial to get all the elements in the extension field $GF(2^m)$, and then computes a random irreducible polynomial of degree t. The primitive polynomials used to generate $GF(2^m)$ are a subset taken from the list compiled by Arash Partow [10].

After creating the Goppa polynomial, it computes matrices X and Y, and multiplies them to get the parity check matrix H. When H is computed, we convert it to a binary matrix of dimension $mt \times n$ and then to its systematic form using the Gauss-Jordan elimination. Lastly, the generator matrix G is computed as follows:

$$G = \left[A^T | I_k\right],$$

where A is taken from H in its systematic form, note that:

$$H = \left[I_{mt} | A\right].$$

Some care should be taken when computing the parity-check matrix as $H = XY$, in particular, the built-in matrix multiplication from Julia should not be used because the entries are in a finite field. Hence, this operation was done manually to ensure that the entries remain in the finite field, but this made matrix multiplication considerably slower.

Once it has an irreducible binary Goppa code, the implementation computes a random permutation matrix P by randomly permuting an identity

matrix, and also a scrambler matrix S along with its inverse. It then proceeds to compute the public generator matrix.

Matrix inversion in $GF(2)$ was a challenge to do efficiently, becoming a major bottleneck in the process. To cut its cost, the Nemo package was used. It is able to efficiently compute the inverse of a matrix in $GF(2)$ with a small overhead caused by the conversion necessary between Julia's built-in unsigned integer type and Nemo's $GF(2)$ type.

4.2 Encryption

Encryption is the most straightforward part of the implementation. It takes a string and converts it to an array of bits, adding padding if necessary. Padding is done by adding zeros to the array of bits until is has a length divisible by k.

It then codes the bit array by parts, using the public key and adding a random binary vector of weight t to every subarray of length k. The encryption function returns an array of ciphertexts.

4.3 Decryption

To decrypt arrays of noisy codewords, the implementation decodes one word at a time and then multiplies the result with $G_J S^{-1}$, where J is the information set. The inverses were computed at the time of decryption, making it slow, so this part of the process was moved to key generation and instead of saving G and S, only $G_J S^{-1}$ was kept.

The first step in the decryption process is to compute the syndrome of a received ciphertext. This task was accomplished by using the simpler formulation of the syndrome presented in this report: the implementation iterates over the ciphertext and the code support, and computes the inverse of $X + \gamma_i$ only when $c_i = 1$.

After the syndrome is computed, we need to get the square root of $T(X) + X \mod g(X)$. Using the method described in Biswas' thesis [2], the square root can be computed efficiently.

Then, we can actually compute the error locator polynomial using a modified version of the extended Euclidean algorithm that halts on the first iteration that the residue's degree falls below $(t + 1)/2$. This yields α and β that we need to compute $\sigma_e(X)$. The implementation makes the polynomial monic after adding its square and non-square parts.

Lastly, the error locator polynomial is evaluated on every element of the code support to find the positions of the errors. Then, the positions of the errors are saved and finally the bits in those positions are flipped. Before decryption returns the obtained plaintext, the implementation checks that the weight of $(plaintext * G) + ciphertext$ is t, and rejects the ciphertext as invalid otherwise [1].

Additional checks to the corrected ciphertexts were implemented to discriminate against messages that do not contain exactly t errors, as well as a test to check that trying to decrypt random vectors always terminates. This last test was performed to four sets of parameters, the results are given in Table 1. None of the random vectors was a valid ciphertext, they all got rejected by the program. This serves as a partial evidence to the non-falsifiability of the McEliece ciphertext assumption put forward in [1]. Note that the last parameter set is "mceliece6960119" - one of those recommended by the "Classic McEliece" proposal [3] to the NIST standardization competition [8].

Table 1: Number of random vectors tested in decryption

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

As a last test, a valid ciphertext was taken and one of the non-error bits was flipped, to see if the program would reject a vector with $t + 1$ errors. Then, the same valid ciphertext was modified in one of the error positions to check if it would get rejected for having $t - 1$ errors. In both cases, the modified ciphertexts were rejected.

At first, almost all functions were pure (Julia makes a copy of mutable structures before passing them as arguments to a function call), but this proved to be too much overhead, so now almost every function uses the original data structures that they receive as arguments.

5 Performance

The generation of keys, encryption, and decryption were timed to measure the performance of the implementation on a Dell OptiPlex 7060 desktop

PC with an Intel Core i7-8700 CPU at 3.20GHz. All the generated keys were tested by encrypting and decrypting the string "Hello, world!". The dimension k of the generated Goppa codes was always much bigger than the binary representation of the string, the message was therefore encoded as a k-bit binary string with a respective padding to fit in one ciphertext. The results are given in Table 2.

Table 2: Mean times for key generation, encryption, and decryption

Parameters (n, k, m, t)	Generation of keys	Encryption	Decryption
(2048, 1707, 11, 31)	62s	0.57ms	1.4s
(2048, 1487, 11, 51)	109s	0.45ms	2.9s
(2048, 1157, 11, 81)	171s	0.32ms	5.9s
(4096, 3604, 12, 41)	312s	3.21ms	4.8s
(4096, 3352, 12, 62)	601s	3.26ms	8.2s
(4096, 2884, 12, 101)	552s	2.84ms	15.7s
(6960, 5413, 13, 119)	1550s	19.60ms	36.5s

Key generation takes more time to finish as the length of the code support and the degree of the Goppa polynomial increase, it also depends on how many times it has to try to generate a parity-check matrix H that can be transformed into its systematic form, and then a generator matrix G and its information set J, something that has probability $\frac{1}{3}$ of succeeding. When this part of the key generation fails, it has to make a new Goppa code and try again. Encryption is the fastest operation of the implementation, but the time it takes to complete still increases significantly when n doubles. Decryption is slower than expected, due to the time it takes to compute the syndrome and the time it takes to evaluate the Goppa polynomial to find the roots of the error locator polynomial. These problems could be addressed by performing fast arithmetic on polynomials using fast Fourier transforms, and by factoring the error locator polynomial instead of finding the roots by direct search.

6 Conclusion

An implementation of the McEliece cryptosystem with a plaintext check in Julia was presented. It performs quite slowly when generating keys and decrypting since it is now in a proof-of-concept stage, and some optimizations

(at both algorithmic and software levels) are not implemented, but it can perform encryption quite fast. The implementation also has additional checks that can test the validity of a given ciphertext. A few tests were done on some sets of parameters to ensure that the decryption process always terminates correctly, such as trying to decrypt random vectors and adding extra errors to a valid ciphertext.

References

[1] F. Aguirre and K. Morozov. "On IND-CCA1 Security of Randomized McEliece Encryption in the Standard Model". In: *Proc. 7th Code-Based Cryptography Workshop (CBC 2019)* (May 18–19, 2019). Darmstadt, Germany.

[2] Bhaskar Biswas. "Implementational aspects of code-based cryptography". PhD Thesis. L'École Polytechnique, Oct. 1, 2010. 61 pp.

[3] *Classic McEliece*. Dec. 12, 2017. URL: `https://classic.mceliece.org/` (visited on 07/23/2019).

[4] D. Engelbert, R. Overbeck, and A. Schmidt. "A Summary of McEliece-Type Cryptosystems and Their Security". In: *Journal of Mathematical Cryptology JMC* 1 (2 2007), pp. 151–199. DOI: `https://doi.org/10.1515/JMC.2007.009`. (Visited on 07/20/2019).

[5] Michiel Marcus. *White Paper on McEliece with Binary Goppa Codes*. Eindhoven University of Technology, Feb. 2019. 35 pp. URL: `https://www.hyperelliptic.org/tanja/students/m_marcus/whitepaper.pdf` (visited on 07/20/2019).

[6] R.J. McEliece. *A Public-Key Cryptosystem Based On Algebraic Coding Theory*. Deep Space Network Progress Report 42-44. 1978, pp. 114–116. URL: `https://ipnpr.jpl.nasa.gov/progress_report2/42-44/44N.PDF` (visited on 07/20/2019).

[7] K. Morozov. "Code-Based Public-Key Encryption". In: R. Nishi. *A Mathematical Approach to Research Problems of Science and Technology*. ed. Vol. 5. Mathematics for Industry. 2014, pp. 47–55.

[8] NIST. *Post-Quantum Cryptography. Round 2 Submissions*. Jan. 30, 2019. URL: `https://csrc.nist.gov/Projects/Post-Quantum-Cryptography/Round-2-Submissions` (visited on 07/20/2019).

[9] R. Overbeck and N. Sendrier. “Code-based cryptography”. In: D.J. Bernstein, J. Buchmann, and E. Dahmen. *Post-Quantum Cryptography*. eds. Springer, 2009, pp. 95–145.

[10] Arash Partow. *Primitive Polynomial List*. URL: `https://www.partow.net/programming/polynomials/index.html` (visited on 07/18/2019).

[11] Ashley Valentijn. *Goppa Codes and Their Use in the McEliece Cryptosystems*. Syracuse University Honors Program Capstone Projects 845. 2015. 41 pp. URL: `https://surface.syr.edu/honors_capstone/845` (visited on 07/20/2019).