# Post Quantum Cryptosystems

Jarrett Thompson

June 20th 2022

**Abstract**

Quantum resistant cryptosystems is a method of encrypting messages which cannot be cracked by a quantum computer. Here we take a look at two quantum resistant cryptosystems that are the round three finilists of the NIST post quantum cryptography challenge.[1] First we will look to see how code based cryptosystem will work called the McEliece cryptosystem. In the Second part we will take a look at NTRU cryptosystem which is lattice-based.

# I. Code Based Cryptosystems

Error correcting codes is the fundamental part of understanding code-based cryptosystems. Before we move on to talk about the McEliece cryptosystem we must first understand error correction. Error correction code is a type of code block that can detect and correct errors within itself This is useful over noisy channels of communication as well as security in getting the correct message. The McElicece uses this because the message being sent is code based which produces the possibility of it being tampered with. To better understand error correction algorithms lets take a close look at how they work using hamming codes.

## 1. Hamming Codes Introduction

Hamming codes were invented by Richard Hamming in 1950 to automatically correct errors in punch card readers. A parity bit is an added bit that helps detect errors within the code. Hamming codes are usually sent in blocks due to the nature of an algorithm. A block of size $2^r - 1$ can contain a message $2^r - r - 1$ bits long. Thus a large message can be sent with just a fraction of parity pits. However hamming codes can only correct one bit errors per block, and if extended, are able to detect two bit errors but not identify and correct them

### Walkthrough

Say we wanted to send this message:

$$1 \quad 0 \quad 0 \quad 1 \quad 1 \quad 0 \quad 1 \quad 0$$

We must first add parity bits to every position to the power of two minus one (Let the starting bit be position 0).

$$p_0 \quad p_1 \quad 1 \quad p_2 \quad 0 \quad 0 \quad 1 \quad p_3 \quad 1 \quad 0 \quad 1 \quad 0$$

Each Parity bit is calculated as follows: $p_0$ is determined by the number of one's in every other 1 bit

$$p_0 \quad p_1 \quad 1 \quad p_2 \quad 0 \quad 0 \quad 1 \quad p_3 \quad 1 \quad 0 \quad 1 \quad 0$$

There are 4 one's thus the value of $p_0$ is 0.
$p_1$ is determined by the number of one's in every other 2 bits

$$0 \quad p_1 \quad 1 \quad p_2 \quad 0 \quad 0 \quad 1 \quad p_3 \quad 1 \quad 0 \quad 1 \quad 0$$

There are 3 one's thus the value of $p_1$ is 1.
Here's a chart showing calculations for the rest of the bits

| Bit position | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Encoded data bits | | p1 | p2 | d1 | p4 | d2 | d3 | d4 | p8 | d5 | d6 | d7 | d8 | d9 | d10 | d11 | p16 | d12 | d13 | d14 | d15 | |
| Parity bit coverage | p1 | ✓ | | ✓ | | ✓ | | ✓ | | ✓ | | ✓ | | ✓ | | ✓ | | ✓ | | ✓ | | ... |
| | p2 | | ✓ | ✓ | | | ✓ | ✓ | | | ✓ | ✓ | | | ✓ | ✓ | | | ✓ | ✓ | | |
| | p4 | | | | ✓ | ✓ | ✓ | ✓ | | | | | ✓ | ✓ | ✓ | ✓ | | | | | ✓ | |
| | p8 | | | | | | | | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | | | | | | |
| | p16 | | | | | | | | | | | | | | | | ✓ | ✓ | ✓ | ✓ | ✓ | |

Notice how each bit has a unique set of parity checks

After filling in the parity bits we get our completed block

$$0 \quad 1 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \quad 0 \quad 1 \quad 0 \quad 1 \quad 0$$

Decoding a hamming code is quite simple. You just take the bits that are non parity bit spaces which is pretty self explanatory. The point of the parity bits is to be able to detect and correct an error. Let's flip a random bit. Here's our new block:

$$0 \quad 1 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \quad 0 \quad 1 \quad 1 \quad 1 \quad 0$$

To start decoding we must make the parity checks again and establish if the paritys are correct or wrong. For $p_0$ you can see that it is 0 so it should have an even number of ones. There is an even number of 1's so it is correct. We are going to keep this in mind by creating a sequence and writing a "0" at the beginning. Next parity bit we check $p_1$ should have an odd number of one's. However it does not, it has 4 one's so it is even. We keep track of this by adding 1 to the beginning of the sequence which is now "10". We keep going to see $p_2$ is correct, and $p_3$ is wrong. and get our sequence "1010". Magically converting "1010" from binary to decimal is 10 which happens to be the position of the bit that had been changed. ($b_9$ if we start our indices at 0). Flip that bit, and take the non-parity bits to return your final message.

### Short Proof

Here is a short proof that helps explain how this algorithm works. Lets say the error lands on a non-parity bit $b_e$. First its important to recognize the way parity bits are set up. A good visualization of this is shown in the figure above. Each time a parity bit $p_n$ is added, it checks the every other $2^n$ bits, creating a unique set of parity checks for each bit.

If the error lands on the parity bit $p_e$, The only parity bit check that fails is $p_e$, because no other parity bits check that bit. The sequence will haveonly a single one in position $e$, thus pointing where the error is which is that parity bit.

If the error lands on non parity bit $b_e$, the sequence will consist of the highist parity by smaller than $e$, so there will be a 1 in position $e$ of the sequence. let this parity bit be $p_r$ Next we subtract $e - 2^r$ to figure out where our next 1 will be in the sequence.

## Hamming Code Extended

Hamming codes are represented in the form of Hamming(n,m) where n is the total number of bits, and m is the non parity, or data, bits So Hamming(7,4) is a 7 bit code block with 4 data bits and most commonly found.

Hamming codes can be extended to add an extra parity bit which can detect and correct up to two errors. This is called SECDED (single error correction, double error correction). This is done by adding a parity bit at the end of the block which checks all bits, including parity bits, as well. It is relatively easy, however, we will not use this but could be of it interest to those who want to implement it.

### Code

When choosing to implement hamming codes, to optimize storage space it is crucial to calculate the risk involved. Two errors in the same code block will result in memory corruption. This can be calculated in by minimizing

$$\frac{\#\ of\ data\ bits}{\#\ of\ total\ bits} < acceptable\ error\ percentage\%$$

In this section we will be using Hamming(7,4).

```
# A comment
x = [5, 7, 10]
y = 0

for num in x:
    y += num

print(y)
```

## 2. The McElicece Cryptosystem

In 1978 a mathematician, Robert McElicece, created a cryptosystem which involves error correction. Today, it is a round three finilist for NIST's post

quantum cryptograpgy standardization process, which means that it could be a standard utility for the future that is uncrackable by quantum computers. For cryptography it was the first method that uses randomness, through error correcting codes, in the encryption process.

## Binary Goppa Codes

Like hamming codes, binary goppa codes cand detect and correct errors however far more powerful. They are the most essential, and magical, part of this cryptosystem. They are much more complex but can detect up to 50 errors in a 1024 bit code block and efficiently decoded. We will not go in depth into this concept but its based on the same principles of error correction code. Because they can detect and decode many errors, we can encrypt the message by adding random errors purposefully. Thus much it far harder for a cyber attack, and every time the message is sent it can be randomly different.

## Creating the Public Key

In cryptography the most common example used during explanations is when Alice wants to send a message to Bob and Eve tries to crack that message. And usually there is a critical piece of information which may need to be shared to Alice and Bob, but not the attacker Eve. The beauty of this system is that the only information that needs to be kept secret are the building block matrices of the public key. The three matrices needed are to build the public key and remain secret to the Bob are:

1. G: A goppa code generator matrix *(n x k)*
2. P: A random permutation matrix *(n x n)*
3. S: A random invertible matrix *(k x k)*

The public key*(n x k)* is created by

$$\hat{G} = S * G * P$$

## Creating the Message

Alice wants to send message m to Bob. She needs two things:

1. m: A message matrix *(1 x n)*
2. e: A random error matrix *(1 x k)*

$$encrypted\ message = m * \hat{G} + e$$

Now Alice can send her message to Bob

### Decryption Method

The decrpytion process are as follows *(Assume calculations are in binary)*:

$$encrypted\ message = m * \hat{G} + e$$

$$encrypted\ message = m * SGP + e$$

first multiply by $P^{-1}$ to on the right

$$encrypted\ message = (m * SGP + e)P^{-1}$$

$$encrypted\ message = m * SGPP^{-1} + eP^{-1}$$

The Goppa Code can correct t errors. Use a Goppa Decoding algorithm to correct the errors.

$$encrypted\ message = m * S$$

Multiply by the inverse of S to get the message.

$$encrypted\ message = m * SS^{-1}$$

$$encrypted\ message = m$$

### Summary

Goppa codes are a very interesting way of showing how we can use randomness in our encryption and decryption methods. Although it will most likely not be determined the standard cryptosystem by NIST for the future, due to its slower speeds and high memomry usage, it's a good introduction into the world of post quantum cryptography. I hope this paper is one step close to helping others appreciate the beauty and ingeniuty of mathematics intertwined with computer science.

## II. References


1. `https://csrc.nist.gov/Projects/post-quantum-cryptography/round-3-submissions`
2. `https://en.wikipedia.org/wiki/Hamming_code`
3. `https://en.wikipedia.org/wiki/McEliece_cryptosystem`
4. `https://en.wikipedia.org/wiki/Goppa_code`